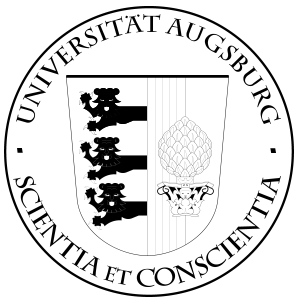


Self-healing distributed systems

Dissertation

for the degree of
Doctor of Natural Sciences (Dr. rer. nat.)

submitted to the
Department of Computer Science
of the University of Augsburg

presented by

Benjamin Satzger

in 2008

Examiner: Prof. Dr. rer. nat. Theo Ungerer
Co-examiner: Prof. Dr. rer. nat. Bernhard Bauer

Date of oral examination: 2008-12-18

Abstract

The growing complexity of distributed systems demands for new ways of control. This work addresses self-healing in distributed environments. The term *self-healing* represents a quite new area of research and is used in a fairly broad way, but can be seen as dynamic fault tolerance. This work proposes generic concepts and algorithms to build self-healing systems.

The detection of node failures in distributed environments is a non-trivial problem. Failure detectors are an important component of many fault tolerant distributed systems. In this work a new failure detection algorithm is proposed with noteworthy features like a high flexibility and good performance. Furthermore an approach is presented to save the message overhead of failure detectors.

New grouping algorithms are introduced in this work to enable a scalable self-monitoring property. This allows an autonomous installation of monitoring relations in complex large scale distributed systems.

A failure recovery engine based on automated planning, which manages a distributed system according to user-defined objectives, is proposed. It is able to generate and execute plans to autonomously recover a system from unwanted states.

Finally, ideas for a generic self-healing architecture for highly complex distributed systems are presented. The design is based on psychological and sociological concepts.

Zusammenfassung

Aufgrund der zunehmenden Komplexität verteilter Systeme werden neue Steuerungs- und Administrierungsmethodiken benötigt. Die vorliegende Arbeit befasst sich mit der Thematik der Selbstheilung in verteilten Umgebungen. Der Begriff *Selbstheilung* stellt einen relativ neuen Forschungsbereich dar und wird thematisch breit benutzt, kann jedoch als dynamische Fehlertoleranz aufgefasst werden. Diese Arbeit schlägt generische Konzepte zur Erstellung selbstheilender Systeme vor.

Das Erkennen von Knotenausfällen in verteilten Systemen ist ein nichttriviales Problem. Fehlerdetektoren sind eine wichtige Komponente vieler fehlertoleranter verteilter Systeme. Diese Arbeit führt einen neuen, besonders flexiblen Fehlerdetektionsalgorithmus mit guten Erkennunsraten ein. Zusätzlich wird ein Ansatz präsentiert, der den Einsatz von Fehlerdetektoren effizienter gestaltet.

Es werden neue Gruppierungsalgorithmen eingeführt, die eine skalierbare Selbstüberwachung ermöglichen und Überwachungsbeziehungen autonom aufbauen.

Eine Fehlerbehebungskomponente basierend auf einem automatischen Planungsansatz wird vorgestellt, die ein verteiltes System gemäß benutzerdefinierter Ziele verwaltet. Sie ist in der Lage, Pläne zu generieren und auszuführen, um selbständig einen spezifizierten Systemzustand wiederherzustellen.

Den Abschluss dieser Arbeit bilden Ideen einer generischen Architektur für hochkomplexe selbstheilende Systeme, basierend auf psychologischen und soziologischen Konzepten.

Acknowledgements

First, I would like to thank my adviser Prof. Dr. Theo Ungerer for his excellent mentoring. I have not only learnt about computer science, but also about performing research, organising things, and communicating with others.

I would also like to thank my co-chair Prof. Dr. Bernhard Bauer for his dedication to my writing and Prof. Dr. Elisabeth André for accepting my request for being examiner.

This is a great opportunity to thank all my colleagues at the University of Augsburg for their support, discussions, and comments on my work.

I would like to thank all members of the Organic Computing research community for inspiring me with their excellent research.

I am grateful for my sources of funding: the priority program 1183 "Organic Computing" of the German Research Foundation (DFG) and the Bavarian state government.

Finally, I wish to thank my wife Melanie Lucas-Satzger who is a great help in anything but computer science.

Dedicated to my family

Contents

1 Introduction

The complexity of today's computer systems is steadily rising. Especially distributed systems interconnect growing numbers of more and more complex heterogeneous devices. IBM has identified this trend which they call *complexity crisis* as one of the major obstacles for the progress in the IT industry [Hor01]. Therefore, new ways have to be found to manage modern computer systems. This research area has recently gained much attention and is considered as "hot topic" in the industrial and academic sector.

IBM's vision of its initiative *Autonomic Computing* (AC) [Hor01] is to design computing systems which can manage themselves given high level objectives from administrators. The human autonomic nervous system is the inspiration for the term AC, as it is adjusting vital low-level functions such as heart rate and body temperature, allowing our brain to deal with other tasks. The AC initiative introduced the demand on future systems to self-configure, self-heal, self-optimise, and self-protect in order to be manageable.

The *Organic Computing* (OC) initiative [ACE+03], a priority program funded by the German Research Foundation (DFG), also addresses the growing complexity of computing systems as major threat. OC includes the targets of AC, but has its focus not on management of data centres like IBM's initiative but the development of generic control mechanisms for technical systems.

The Recovery-Oriented Computing (ROC) project [PBB+02] is a joint research project of University of California at Berkeley and Stanford University investigating novel techniques for building highly-dependable Internet services. ROC takes the perspective that hardware faults, software

bugs, and operator errors are facts to be coped with, not problems to be solved. A large amount of administration time needs to be dedicated to system failures. A survey of total cost of ownership for cluster-based services [GAKM02] suggests that a third to a half result from repairing failures. Experienced experts able to manage such complex systems are becoming more and more rare. Furthermore, these systems continue to rise in size and complexity while the demands on the availability and reliability of IT services are by no means decreasing.

The solution for the described issues is to make complex systems *self-healing*. This means the ability to detect failures and compensate for them if possible, and adapt to a changing environment as well as to changing objectives. And all this at runtime. Helping to achieve these requirements is the aim of this dissertation.

The next section introduces necessary terms and definitions, Section 1.2 provides an overview of related work. Then, Section 1.3 highlights the challenges for this work and Section 1.4 the contributions and structure of this work. Finally, Section 1.5 gives information about previous publications incorporated into this work.

1.1 Terminology

The terms introduced in this section are mostly based on the work of Avizienis et al. [ALR04] who provide definitions related to dependability.

A *system* is an entity that interacts with its *environment*, i.e. other systems, users, and so on. A *functional specification* describes what a system is intended to do, while the *behaviour* of a system describes what the system is actually doing. A system provides *correct service* if it is behaving according to the functional specification. A deviation of a service from the specification is called *service failure* or just *failure*. The port of the state of a system that may lead to its subsequent system failure is called *error*. The cause of an error is a *fault*. A fault is active when it causes an error, otherwise it is dormant.

The *dependability* of a system is the ability to avoid failures which are unacceptable for users. Some attributes are [ALR04]:

Availability: Readiness for correct service.

Reliability: Continuity of correct service.

Safety: Absence of catastrophic consequences on the users and the environment.

According to [ALR04] there are four means to attain dependability:

Fault prevention: Prevent the occurrence or introduction of faults.

Fault tolerance: Avoid failures in the presence of faults.

Fault removal: Reduce the number and severity of faults.

Fault forecasting: Estimate the present number, the future incidence, and the likely consequences of faults.

In contrast to the definitions introduced so far, the term "self-healing" stands for a quite new area of research and is used rather broadly. In this work it is argued that self-healing deals with fault tolerance for dynamic systems. Therefore, it is necessary to detect incorrect service and to find ways to repair the system. Self-healing implies that this is performed mainly automatically and autonomously by the system itself. Gosh et al. [GSRU07] define self-healing systems as being able to perceive that it is not working correctly and to make the necessary adjustments to restore itself. Self-healing systems should be able to dynamically adapt their behaviour in response to changes in their environment. At runtime, they should have the ability to deal with situations not anticipated in design-time. Important aspects for self-healing capabilities in distributed systems are fault/failure detection by run-time monitoring, planning and execution of repair actions, and both regarding scalability issues.

1.2 Related work

Gosh et al. [GSRU07] argue that most self-healing concepts are still in their infancy. The research labelled with the term "self-healing" has various facets and applications areas. Furthermore, systems and methods which could be called "self-healing" do not stress this term. The aim of the following is to report on selected research applying interesting techniques for self-healing, reconfiguration, or adaptation, and on reference architectures.

Figure 1.1 shows the AC reference architecture as proposed by IBM [IBM06]. The lowest layer contains the system components, or managed resources. The next layer incorporates consistent, standard manageability interfaces for accessing and controlling the managed resources. Layers three and four automate some portion of the IT process using an autonomic manager. These managers implement a control loop to enable self-configuration, self-healing, self-optimisation, and self-protection. They can be orchestrated to deliver system wide autonomic capability.

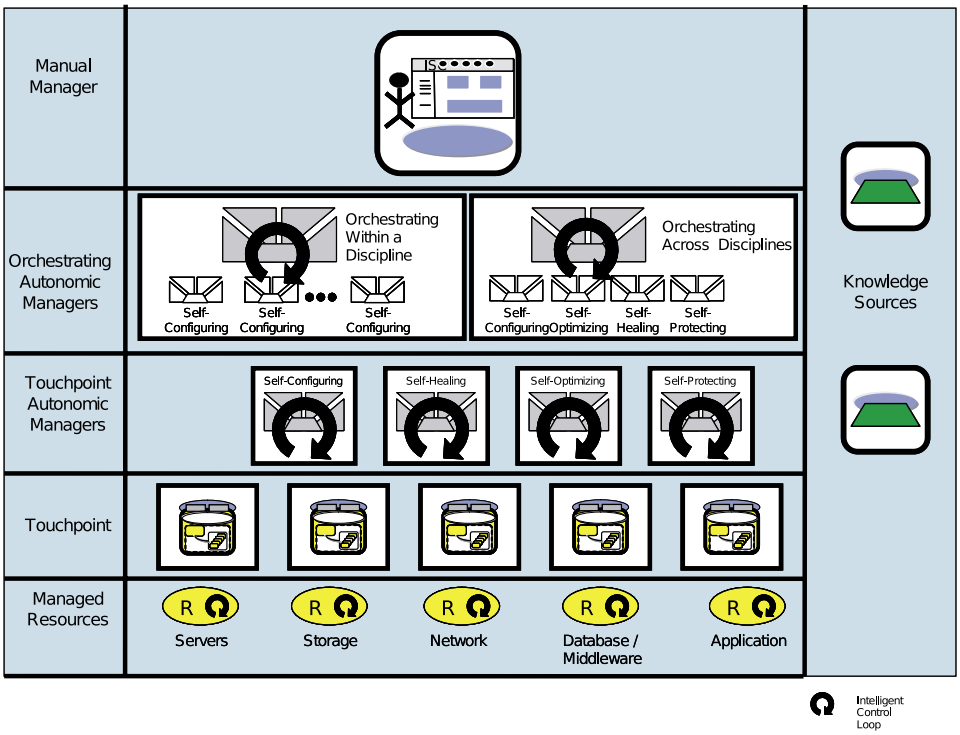


Figure 1.1: *Autonomic computing reference architecture*

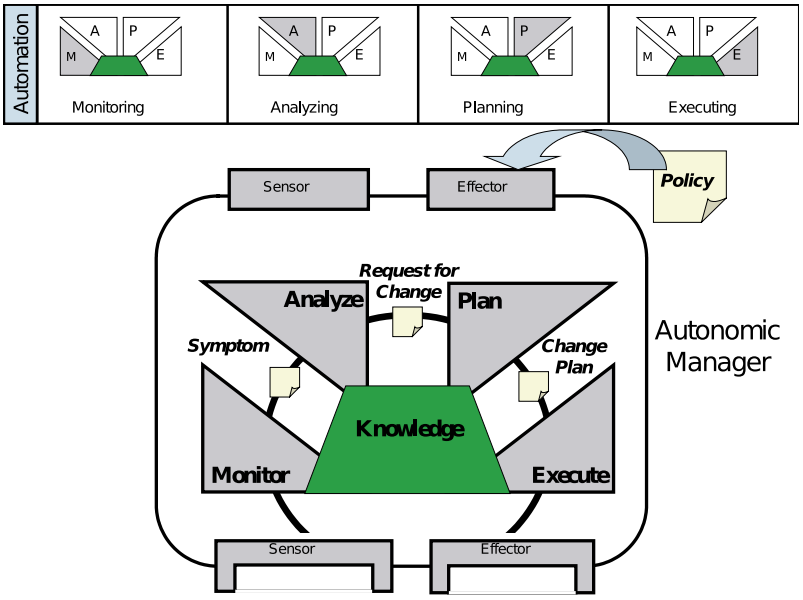


Figure 1.2: MAPE *cycle of an autonomic manager*

Figure 1.2 shows details of an autonomic manager, mainly the intelligent MAPE loop consisting of the following four steps:

Monitor: Collection, aggregation, and filtering of information.

Analyse: Analysis of the gathered information.

Plan: Construction of actions needed to achieve goals and objectives.

Execute: Execution of such actions.

In [BMMSP06], Branke et al. propose a generic observer/controller architecture for OC, illustrated in Figure 1.3. Its working principle is similar to the

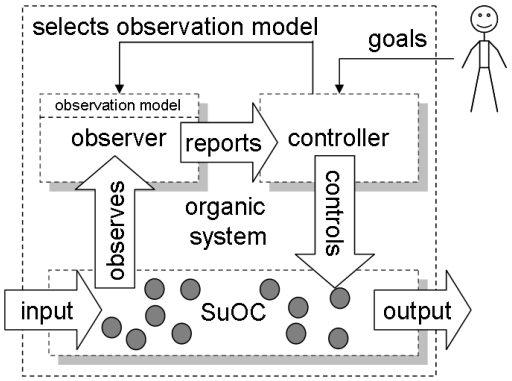


Figure 1.3: *Observer/controller architecture*

MAPE cycle of AC. The observer component monitors the system and thus has the ability to detect flaws. In such cases, the controller takes corrective actions. The system which is managed by such an observer/controller is called *system under observation and control* (SuOC). The user can manipulate the system behaviour by defining goals.

Prothmann et al. [PRT+08] propose a novel architecture for traffic light controllers, shown in Figure 1.4, which have the ability to adapt to the current traffic situation. Their approach can be seen as an instance of the architecture presented in [BMMSP06]. A standard traffic light controller (TLC) is extended by a two-layered observer/controller component that reconfigures the TLC depending on the current traffic. Layer 1 of the observer/controller component is responsible for the selection of TLC parameters based on the monitored data. This is performed by a learning classifier system (LCS) selecting appropriate rules from a rule base. On Layer 2 an offline generation of new rules is performed aiming to optimise the existing rule base.

There are interesting publications in the area of dynamic software architectures. In a position paper [And98], Andersson proposes a technique where rule-based agents monitor the architecture and perform simple reconfigurations. Oreizy et al. [OGT+99] argue that self-adaptive software will provide the key to applications that retain full plasticity throughout their life cycle and that are as easy to modify in the field as they are on the drawing board.

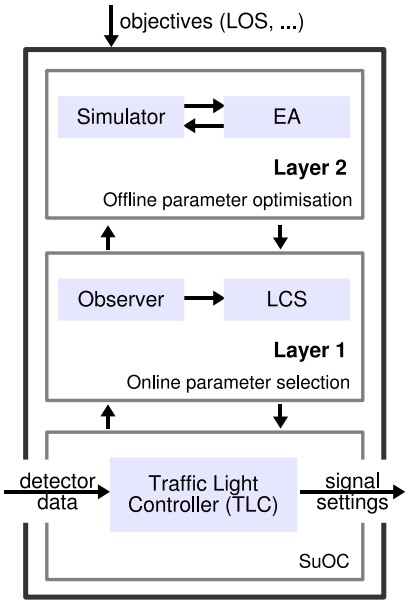


Figure 1.4: *Organic Traffic Control Architecture*

They identify observation, detection of events, monitoring, and planning as vital parts of self-adaptive systems.

1.3 Challenges

The approach to this work is to identify key research challenges related to the construction of self-healing distributed systems and middleware and to propose methods of resolution for them.

The first question addressed is how to detect node failures in distributed systems. In such an environment failed nodes are often indistinguishable from some functional nodes. A failure detector in such a sense is an oracle that can intelligently suspect nodes or processes to have failed. For a runtime failure detection and a consequent healing, failure detection services are needed. Thus, failure detection algorithms deliver necessary monitoring information and form a basis for a self-healing capability in distributed systems.

In complex distributed systems, the issue of scalability is of particular importance for any technique applied there. Therefore, an important aspect for self-healing systems is the formation of nodes into groups to apply a kind of divide and conquer strategy in order to achieve scalability.

Self-healing systems should perform recovery and reconfiguration actions and adapt to changing environments and user needs during runtime. There-

fore, they are in need of more intelligence than traditional fault-tolerant systems in order to find ways to recover a system. This enables systems to autonomously adapt their behaviour to a given functional specification.

1.4 Contributions and outline

The contributions contained in this dissertation can be classified as falling into three areas: detection of node failures, group formation algorithms, and autonomous system recovery. These concepts are an approach towards self-healing distributed systems and middleware with self-healing capabilities addressing the above stated challenges.

In Chapter 2, a new failure detection algorithm is presented. It adapts to changing network conditions and outputs suspicion values within a continuous scale instead of Boolean values. Experiments compare the new algorithm to well-known failure detectors and attest an excellent detection performance at low computational demands. Furthermore, a technique is devised to decrease the message overhead of a class of failure detectors.

Chapter 3 gives a formal problem statement of group formation tailored to self-healing distributed systems. Three instances of such grouping algorithms are proposed. These algorithms are compared regarding their ability to install monitoring relationships within the nodes of a distributed system. Together with failure detectors, this allows a scalable mutual monitoring of nodes even in environments with many nodes.

While the chapters mentioned above mainly deal with components for the observation of a system, in Chapter 4 a generic controlling technique for self-healing systems is presented based on an automated planning approach.

Chapter 5 discusses ideas for a generic self-healing architecture for highly complex distributed systems, mainly based on psychological and sociological concepts. It can be seen as an outline of future work for this thesis.

The dissertation ends with Chapter 6 which summarises this work.

The main chapters 2 to 5 all touch different areas of research and are organised such that they are quite self-explanatory and separately readable.

1.5 Publications

Parts of the contents of this dissertation appear in previous publications:

[SPTU07a] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *SAC 2007: Proceedings of the 22nd ACM symposium on Applied computing*, pages 551–555, New York, NY, USA, 2007. ACM.

[SPTU07b] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. Variations and evaluations of an adaptive accrual failure detector to enable self-healing properties in distributed systems. In *ARCS 2007: Proceedings of the 20th International Conference on Architecture of Computing Systems*, volume 4415 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2007.

[SPTU08a] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A lazy monitoring approach for heartbeat-style failure detectors. In *ARES 2008: Proceedings of the 3rd IEEE International Conference on Availability, Reliability and Security*, IEEE Transactions, pages 404–409. IEEE Computer Society, 2008.

[SPTU08b] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. Using automated planning for trusted self-organising organic computing systems. In *ATC 2008: Proceedings of the 5th International Conference on Autonomic and Trusted Computing*, volume 5060 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2008.

[SU08] Benjamin Satzger and Theo Ungerer. Grouping algorithms for scalable self-monitoring distributed systems. In *Autonomics 2008: Proceedings of the 2nd ACM/ICST International Conference on Autonomic Computing and Communication*, 2008.

Chapter 2 includes material from [SPTU07a], [SPTU07b], and [SPTU08a]. Some basic principles of Chapter 4 are presented in [SPTU08b] and Chapter 3 is largely based on [SU08].

2
Failure detection

2.1 Introduction

The detection of failures in distributed environments is a crucial part for developing dependable, robust, and self-healing systems. The term *failure* in this context refers to a node outage. In this chapter a new failure detection algorithm is introduced which is characterised by flexibility, low overhead, and high detection quality. Instead of outputting whether a component has failed or not, it outputs a level of confidence. These features allow to design generic failure detection services based on the proposed failure detector. Furthermore, some variations of the basic algorithm are introduced to further improve its performance. An evaluation is provided for all algorithms using message delay and loss models of the Internet. The results attest the introduced failure detector a very good detection quality compared to other algorithms, especially in the case of message losses.

Section 2.2 provides a short introduction to distributed systems, Section 2.3 gives an overview of failure detectors and related work. Section 2.4 presents the proposed failure detection algorithm and Section 2.5 introduces some extensions of this basic algorithm. Then, Section 2.6 describes the conducted evaluations. Finally, Section 2.8 concludes the chapter.

2.2 Distributed systems

This work follows the definition of Barbosa [Bar96] who defines a *distributed system* as an interconnected collection of autonomous computers, processes, or processors. To refer to these interconnected elements the most common terms are *node* or *process* - in the context of failure detection *process* is more common. Barbosa calls processes *autonomous* if they are at least equipped with an own private control. Following Flynn's taxonomy [Fly72] a distributed system is generally recognised to be a MIMD architecture. Furthermore, the nodes in a distributed system do not share a single address space but make calls to other address spaces, possibly on other machines [KWWW94].

Basically, information in distributed systems can be exchanged either by *message-passing* or by using a *distributed shared memory* (DSM) [TvS02]. As the message-passing model is more general and DSM systems are normally built upon sending messages, we will consider the information exchange in distributed systems to be based on message-passing.

Thus one can define a distributed system as a set of processes $\Pi = \{p_1, \ldots, p_n\}$ where $p_i \in \Pi$ is able to communicate with $p_j \in \Pi$ if they are connected by a *communication channel*. Unless otherwise noted it is assumed that the communication channels connecting the processes are *bidirectional*. Therefore, if p_i can send messages to p_j then also p_j can send messages to p_i. Because of these assumptions a distributed system can be modelled as a bidirectional graph $G = (V, E)$ where the processes are represented by the vertices and the channels are represented by the edges of G.

The graph of a distributed system is called *network topology*. Some special topologies exist which play an important role in practical systems. Amongst these are:

Star: A *star* denotes a network with one central node that is connected to all other nodes. Let $G = (V, E)$ be the corresponding graph and n the number of vertices. Then the number of edges is $n - 1$ and all edges are connected to the central vertex.

Ring: The graph of a *ring* has n edges and is connected. This means a path between any two nodes exists.

Clique: In a *clique* a communication channel exists between any two nodes. Thus its graph is *complete*.

If not stated otherwise in this work it is assumed that a distributed system is always connected.

2.2.1 Communication

The communication system used in this work is modelled similar to [DLS88]. Each process has a *buffer*. This buffer contains the messages that have been sent to the process, but not yet received. Each process p can perform one of the following two sending primitives:

Send(m**,** q**):** Sends a message m over the communication channel and places it in q's buffer.

Receive: Removes all messages from p's buffer and delivers the messages to p.

The process of sending a message from p to q is illustrated in Figure 2.1.

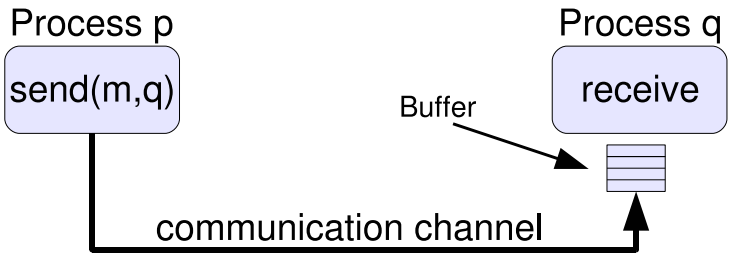


Figure 2.1: *Communication model*

2.2.2 Synchrony

Synchrony describes assumptions of timing aspects. In distributed systems the most important are the time it takes to send a message over a communication channel and the time taken by a processing device to execute a piece of code [LAF99].

The timing assumptions can be classified as follows [DLS88, LAF99]:

Synchronous: A timing attribute is *synchronous* if a fixed upper bound exists that is known a priori.

Asynchronous: A timing attribute is *asynchronous* if no fixed upper bound exists.

Partially synchronous: A timing attribute is *partially synchronous* if it is neither synchronous nor asynchronous.

An example for partial synchrony is a timing attribute with an upper bound that is not known a priori. Another example is a communication system that is delivering messages with a known upper bound, but is unreliable and looses messages sometimes.

Note that different definitions of synchrony exist. Often an operation is called synchronous if it blocks a process till the operation is completed. An asynchronous operation does not block the process and only initiates the operation. But in this work the above introduced meaning of synchrony is applied.

2.2.3 Failure models

Unfortunately, in distributed systems failures can occur. A failure model describes the type of failures that might happen. While there are slight discrepancies in literature regarding their definitions, in this work the following failure models are defined:

Fail-stop: Processes are considered to execute their programs correctly. If a failure happens at a certain time the failed process stops permanently. This models a crash of a process that never recovers. However, in the fail-stop model the failure of a process is detectable. Faulty processes answer that they have crashed when asked.

Crash: Similar to the fail-stop model, but failed processes stop to send any messages and do not indicate their failure.

Crash-recovery: Processes can crash but may recover afterwards to the correct state before crashing.

Crash-omission: The crash-omission model includes the crash model but additionally communication channels may loose messages while sending.

Crash-recovery-omission: This model includes the crash-recovery model with additional message losses of communication channels.

Byzantine: In the Byzantine model no assumptions about the behaviour of a failed process are made. This arbitrary failure model is the most general one and includes denial of service attacks, pretending to be another process, and so on. But failures need not to be meant ill and might just be e.g. a program bug.

The presented models refer to failures on the level of processes. However, a failure of a process does not imply a failure on system level. It is indeed one goal of self-healing systems to avoid failures on lower levels to cause failures on higher levels.

2.3 Taxonomy, definitions, and survey of failure detectors

In this section the taxonomy and definitions are given that are necessary to talk reasonably about failure detectors. Normally, distributed failure detectors are considered in systems with crash failures where each process has access to a local failure detector [CT96]. Each local failure detection module monitors a subset of the processes in the system maintaining a *suspect list*. This is a list of processes that are currently suspected to have crashed.

2.3.1 Unreliability

Several impossibility studies [CHT96, Lyn89, FLP85] show that perfect failure detectors cannot exist in asynchronous distributed systems. The major reason is the impossibility to distinct with certainty whether a process has failed or the communication network is just slow. The most famous impossibility result is related to the *Consensus* problem where each process proposes a value and the processes that do not crash have to agree (termination) on the same value which has to be one of the proposed values (safety). One instance for this problem is the *Transaction commit* problem in distributed database systems [DS82, GM82]. The problem is for all the data manager processes which have participated in the processing of a particular transaction to agree on whether to install the transaction's results in the database or to discard them. The latter action might be necessary if some data managers were unable to carry out the required transaction processing. Whatever decision is made, all data managers must make the same decision in order to preserve the consistency of the database. It has been shown that this apparently simple problem actually has no deterministic solution as soon as even only one process can crash [FLP85].

Chandra et al. [CT96] introduced the idea of failure detectors as an unreliable distributed oracle at which it is possible that (1) a process has failed but is not suspected as well as (2) a process is suspected but has not failed. Moreover a failure detector can change its mind - for example stopping to suspect a process it previously suspected. In consequence, the authors of [CT96] characterise failure detectors by specifying their properties regarding completeness and accuracy. Completeness refers to failure detectors eventually suspecting crashed processes, while accuracy restricts the mistakes that a failure detector can make.

In the following the definitions of the two completeness and four accuracy properties are given [CT96]:

Strong completeness: Eventually every process that crashes is permanently suspected by *every* correct process.

Weak completeness: Eventually every process that crashes is permanently suspected by *some* correct process.

Strong accuracy: No process is suspected before it crashes.

Weak accuracy: Some correct process is never suspected.

Eventual strong accuracy: There is a time after which correct processes are not suspected by any correct process.

Eventual weak accuracy: There is a time after which some correct process is never suspected by any correct process.

Based on their properties regarding completeness and accuracy, failure detectors can be categorised into eight classes. The resulting classes and corresponding notations are given in Table 2.1, according to Chandra et al. [CT96].

Completeness	**Accuracy**			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\Diamond\mathcal{P}$	*Eventually Strong* $\Diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\Diamond\mathcal{Q}$	*Eventually Weak* $\Diamond\mathcal{W}$

Table 2.1: *Classes of failure detectors regarding accuracy and completeness*

It is highly desirable that failure detectors satisfy completeness and accuracy properties because then the above mentioned impossibility results change. It is shown that it is for instance possible to solve *Consensus* using each one of the eight classes of failure detectors [CT96]. Also *Atomic broadcast* [CASD85], another problem that cannot be solved in asynchronous systems without demands on failure detectors, now can be solved.

A broadcast protocol is called *atomic* if it holds the following properties for a time constant Δ, the so-called broadcast termination time [CASD85]:

Atomicity: If any correct process delivers an update at time *U* on its clock, then this update was initiated by some process and is delivered by each correct process at time U on its clock.

Order: All updates delivered by correct processes are delivered in the same order by each correct process.

Termination: Every update whose broadcast is initiated by a correct process at time T on its clock is delivered at all correct processes at time $T + \Delta$ on their clocks.

An atomic broadcast protocol can be used to implement the abstraction of a *synchronous replicated storage*. This denotes a distributed storage where every correct process has the same value for the replicated data. An update of the replicated data is limited by the broadcast termination time Δ.

Another interesting result is that every failure detector that satisfies weak completeness also satisfies strong completeness [CT96]. In asynchronous systems however it is impossible to implement a failure detector that belongs to one of the classes shown in Table 2.1.

2.3.2 Quality of Service

Chen et al. [CTA00] studied the *quality of service* (QoS) of failure detectors. QoS in this context means measures that indicate (1) how fast a failure detector detects actual failures, and (2) how well it avoids false detections. To quantify the QoS, the authors of [CTA00] propose a set of metrics. They divide these metrics into primary metrics and derived metrics:

Primary metrics

Defining the metrics we assume two processes p and q where p is monitoring q.

Detection time (T_D): T_D is the time that elapses from q's crash to the time when q starts to suspect p permanently.

Mistake recurrence time (T_{MR}): The mistake recurrence is the time between two consecutive mistakes. The term mistake stands for the erroneous suspicion of a process.

Mistake duration (T_M): This measures the time it takes the failure detector to correct a mistake.

For applications some other aspects of failure detectors may be interesting, too. That is why Chen et al. [CTA00] propose four other accuracy metrics that are derivable from the primary metrics but provide meaningful information.

Derived metrics

Average mistake rate (λ_M): The average mistake rate is the rate at which a failure detector makes mistakes.

Query accuracy probability (P_A): This denotes the probability that the output of the failure detector at a random time is correct.

Good period duration (T_G): If the failure detector makes no mistakes during a certain period, this period is called *good*. This metric measures the time of a good period.

Forward good period duration (T_{FG}): This represents the time that elapses from a random time at which p trusts q to the next time where the output changes to "suspecting q".

2.3.3 Monitoring strategy

Two main monitoring approaches for failure detectors exist: push and pull.

Assuming process p has a failure detector monitoring q. Using a push failure detector q has to send heartbeat messages to p. This information is used by p to draw conclusions about q's status. A simple failure detection algorithm using the push approach works as follows: q sends heartbeat messages at regular time intervals Δ_i to p. When p receives a heartbeat message it trusts q for a certain period of time Δ_{to}. If this period elapses without receiving a newer heartbeat, p starts to suspect q (see Figure 2.2). Due to the use of heartbeat messages, failure detectors that are based on a push mechanism are also called heartbeat-style failure detectors. An algorithm that uses a push failure detector can be found in [CTA00].

In systems with a pull failure detection (e.g. [HK97]) the monitored node adopts a passive role. p monitors q by sending "are you still alive"-messages every Δ_i. If p does not receive an answer from q within a certain period of time Δ_{to}, p is suspecting q (see Figure 2.3).

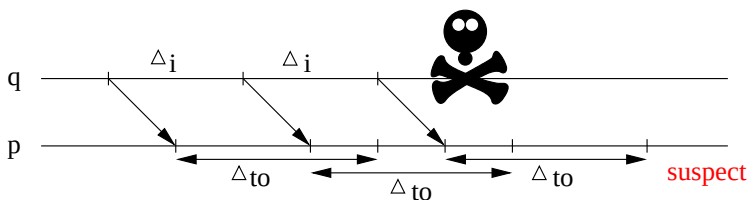


Figure 2.2: *push failure detection*

Failure detectors using the push paradigm have some benefits compared to pull failure detectors. They need only half the messages for an equiva-

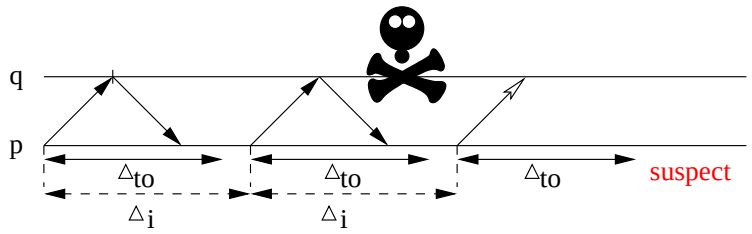


Figure 2.3: *pull failure detection*

lent failure detection quality. Furthermore it is rather hard to determine the timeout Δ_{to} as two messages have to be taken into account which are both sent over the network and subject to network delays.

2.3.4 Adaptiveness

Implementing failure detectors for practical systems, it is a crucial part to determine parameters like Δ_i and Δ_{to}. If for instance the timeout Δ_{to} is short, then failures are detected fast but the probability for false detections is high. A long timeout results in a longer detection time but fewer false detections. These trade-offs have to be considered. Another aspect that should influence the values for the parameters is the underlying network and its speed, latency, bandwidth, load, etcetera.

Adaptive failure detectors [FRT01, CTA00, HDYK04] are able to adjust to changing network conditions. The behaviour of a network can be significantly different during high traffic times as during low traffic times regarding the probability of message loss, the expected delay for message arrivals, and the variance of this delay. Adaptive failure detectors arrange their parameters in order to meet the current conditions of the system. Thus adaptive failure detectors are highly desirable.

Chen et al. [CTA00] propose a well-known adaptive failure detection approach based on a probabilistic analysis of network traffic. The protocol uses sampled arrival times to compute an estimation of the arrival time of the next heartbeat. The timeout is set according to this estimation plus a constant safety margin, and is recomputed after each arrival of a new heartbeat.

Bertier et al. [BMS02] combine Chen's estimation with another estimation developed by Jacobson [Jac88] for a different context. Their approach is similar to Chen's - however they do not use a constant safety margin but compute it with Jacobson's algorithm.

2.3.5 Laziness

Lazy failure detection algorithms [FRT01] try to reduce the networking overhead that arises e.g. from sending heartbeat messages. To do so, they use the application messages of communicating processes in order to save failure detection messages. Only when two processes are not communicating, failure detection messages are used. Thus, if they are exchanging messages frequently the failure detector might have to send no single message. Hence, these algorithms are closely related to the actual communication behaviour of the processes.

In Section 2.7, lazy failure detectors are discussed in more detail and a new approach for lazy monitoring is introduced.

2.3.6 Accrualty

The principle of an accrual failure detector, introduced by Hayashibara et al. [HDYK04], is not to output whether a process is suspected to have crashed or not. Rather they give a suspicion information on a continuous scale while higher values indicate a higher probability that the monitored process has failed.

Hayashibara et al. motivate the benefits of accrual failure detectors over conventional Boolean failure detectors. As principal merit they indicate that accrual failure detectors favour a nearly complete decoupling between application requirements and the monitoring environment. As accrual failure detectors output a suspicion probability information, it is left to the application to interpret this information and choose appropriate actions. To illustrate the benefits of accrual failure detectors they quote an example of an application with one master and some working processes. The master assigns jobs to the workers and holds a list of available worker processes. If workers are idle, it sends them new jobs and collects the results after computation. Assuming that the workers might crash, the master has to be able to detect this and take appropriate actions, otherwise the system might block forever. With an accrual failure detector some low crash probability could be set to stop sending jobs to this worker when this probability is reached. When another higher probability is reached the master could cancel all unfinished jobs that have been assigned to this worker and resubmit them to other workers. When reaching a high probability, then the master could finally remove the process from the list of available workers. To implement this behaviour traditional failure detectors are inappropriate.

Hayashibara et al. [HDYK04] propose a so-called φ failure detector that is based on an estimation of inter-arrival times of heartbeats assuming that

inter-arrivals follow a normal distribution. A suspicion value is basically computed as follows: The monitoring process stores the receipt times of the heartbeats in a sampling window of a fixed size. Based on this data amongst others the mean and variance of heartbeat inter-arrival times are computed. The authors assume the inter-arrival times following a normal distribution and use this information to compute the probability that starting from now another heartbeat will arrive. This probability is transformed using a logarithmic function and the result is then outputted as suspicion value.

2.4 The failure detector

In this section a new failure detection algorithm that can be described as an adaptive accrual algorithm is presented. It has been designed for flexible generic usage as a basis to realise failure detection services. After the description of the algorithm that is given in the following, Section 2.5 discusses extensions and variations of it. Then, the evaluation results are presented in Section 2.6.

2.4.1 Design

A distributed system consisting of a set of n processes $\Pi = \{p_1, p_2, \ldots, p_n\}$ is considered. Each pair of processes is connected by a communication channel that can be used to send and receive messages. Communication primitives are assumed as described in 2.2.1.

The failure detector uses a push-style monitoring approach, i.e. the monitored processes have to send heartbeat messages to the monitoring processes every time step Δ_i. Failure detectors using the push paradigm have some benefits compared to pull failure detectors which have been cited in the corresponding part in Section 2.3. The heartbeats are sent in uniform intervals, the shorter the interval the better the detection quality but the higher the network and computation overhead. Different intervals can be used for different processes to allow for more important services to be monitored closer and to detect failures faster. The effects of the lengths of the heartbeat intervals should become clearer within the rest of this chapter.

Whenever a heartbeat message from a process is received by the failure detector it stores the time that has elapsed from the arrival of the last heartbeat message. This time is called *inter-arrival* time and its value is stored in a *sampling window*. A sampling window typically has a fixed size; if the latest heartbeat inter-arrival time exceeds this size limitation the oldest sample is removed in order to insert the latest one. The size of the sampling win-

dow has a strong effect on the behaviour of the failure detector. A small sampling window causes the failure detector to adapt fast to changing conditions, whereas a big sampling window results in the failure detector to factor long term experience into the failure detection process. A separate sampling window for each monitored process exists, containing the respective heartbeat inter-arrival times.

The monitoring process, i.e. the process that has access to the failure detector, has nothing to do with the monitoring at all. It only has to be aware of a simple interface. The monitoring process can ask the failure detector for an approximation of the probability $P_{fail}(x)$ that a certain monitored process x has failed whereupon a probability value within $[0, 1]$ is returned. This value represents the probability that the monitored process has failed based on the observations made by the failure detector so far.

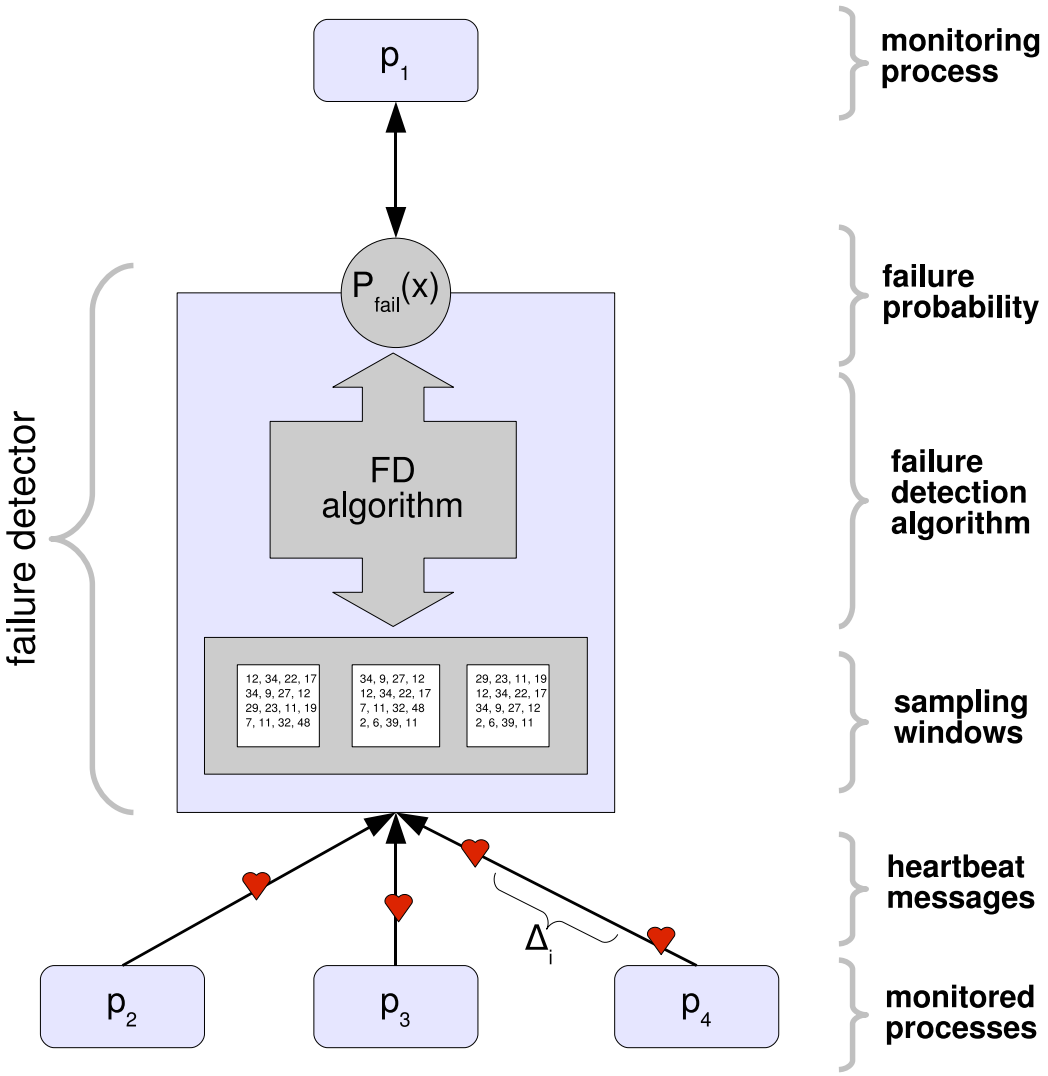


Figure 2.4: *Design of the failure detector*

The design of the failure detector is illustrated in Figure 2.4 which shows a process p_1 monitoring p_2, p_3, and p_4. The monitored processes have to send heartbeats in certain intervals Δ_i to p_1. The failure detector of p_1 maintains a sampling window for each monitored process containing the corresponding heartbeat inter-arrival times. The failure detector offers an interface to p_1 returning the current failure probability for the particular process. The failure detection algorithm calculates a failure probability based on the values in the sampling windows.

Having introduced the basic features of the failure detector, in the following, the underlying algorithm is explained.

2.4.2 Basic idea of the algorithm

The basic concepts will be made clear with an example: Assuming two processes, p and q, p is monitoring q, and q sends heartbeat messages to p every $\Delta_i = 1$ second. Process p[1] manages a sampling window S with information about the inter-arrival times of the last 1000 heartbeats it received. At a certain point during runtime $S = [1.083s, 0.968s, 1.062s, 0.993s, 0.942s, 2.037s, 0.872s, \ldots]$. Furthermore p stores the time of the last received heartbeat called *freshness point* f. Based on the sampled inter-arrival times and f the algorithm estimates the probability that q has failed. Figure 2.5 shows the values of S as a histogram.

The shape of the histogram depends mainly on Δ_i and the communication channel connecting q and p. In this particular example Δ_i is one second. The communication channel has a message loss rate of 10% and a certain fluctuating message sending delay. The peak at two seconds arises from one lost heartbeat message, the peak at three seconds arises from two consecutive lost heartbeat messages, and so forth.

This histogram can be seen as an approximation of the *probability density function* of the distribution of the inter-arrival times. Based on this histogram the cumulative frequencies of the values in S are easily computable. The cumulative frequencies in turn can be seen as an approximation of the corresponding *cumulative distribution function*. A cumulative distribution function (CDF) completely describes the probability distribution of a real-valued random variable, in our case the inter-arrival times of the heartbeat messages. The CDF $F(t_\Delta) = P(X \leq t_\Delta)$ represents the probability that an inter-arrival time takes on a value less than or equal to t_Δ. The values of the CDF are also a reasonable indicator for the crash of q: Assume p is waiting since time t for the next heartbeat from q. $F(t_\Delta) = x$ means "p is waiting

[1] "Process p" is used synonymously to "process p's failure detector" for a shorter nomenclature.

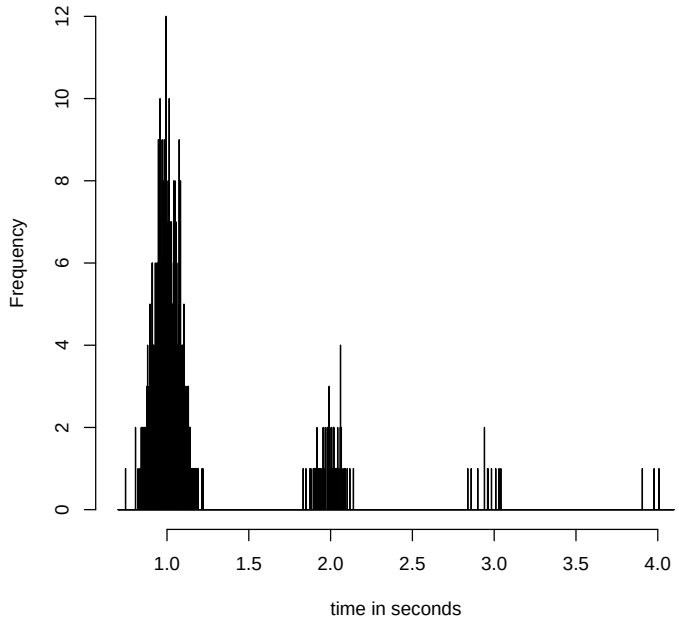


Figure 2.5: *Histogram of the sampled inter-arrival times in S*

for the next heartbeat message since t_Δ seconds and the probability that no further heartbeat message arrives is x''. The longer p is waiting for the next heartbeat the higher are the values of F.

Figure 2.6 shows the cumulative frequencies of the values in S. We use these cumulative frequencies as an estimation of the real CDF of the inter-arrival times and to compute a suspicion value for the failure of q.

Finally, the mathematical function P_{fail}, the failure detector uses to compute a suspicion value for q, is specified. It represents an estimation of the CDF of the values in S. The function simply computes the percentage of elements in S that are smaller than or equal to t_Δ. It is easy to see that this function generates a graph like in Figure 2.6, given the above histogram.

$$P_{fail}(t_\Delta) = \frac{|S^{t_\Delta}|}{|S|} \tag{2.1}$$

where

- $|S|$ is the number of elements in S,
- t_Δ is the time that has elapsed since the last freshness point
- $S^{t_\Delta} = \{x \in S \mid x \leq t_\Delta\}$
- $|S^{t_\Delta}|$ is the number of elements in S^{t_Δ}

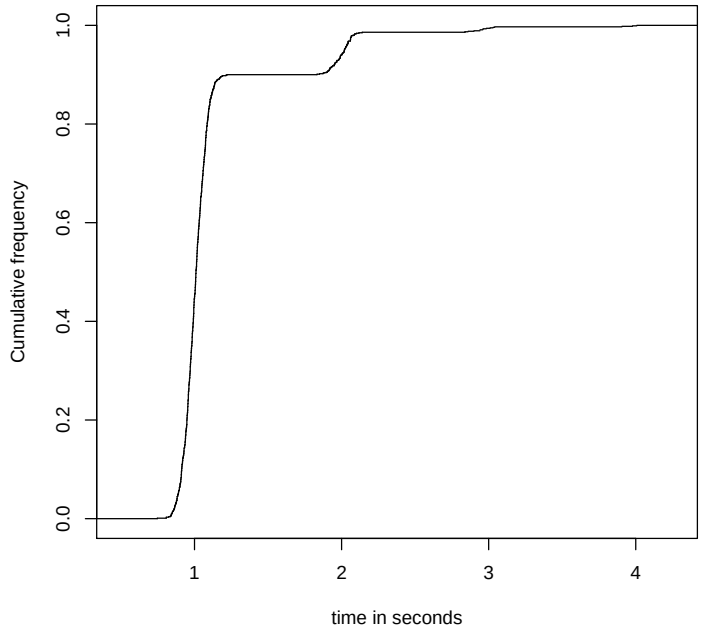


Figure 2.6: *Cumulative frequencies of the sampled inter-arrival times in S. An approximation of the CDF by P_{fail}.*

2.4.3 The algorithm in detail

Algorithm 1 shows the failure detection algorithm in detail. Again two processes p and q are considered where q is monitored by p and its only task is to send heartbeat messages to p every Δ_i seconds. p's failure detector has the variables f, S, and η. f is the freshness point - in our case always the time when p received the last heartbeat message. S is the list of the heartbeat message inter-arrival times. η is the maximal size of S - the so called sample window size. Whenever p receives a heartbeat, the failure detector appends $t_\Delta = t - f$ (t is the current time) to S and sets $f = t$ afterwards. p removes the head of S if the list grows to a size of $\eta + 1$ in consequence of appending the latest inter-arrival time.

If p wants to know the current suspicion value of q, the failure detector counts the number of elements in S^{t_Δ} ($S^{t_\Delta} = \{x \in S \mid x \leq t_\Delta\}$) and returns its normalised value $\frac{|S^{t_\Delta}|}{|S|}$, where $|S|$ is the current size of S. This computation of the suspicion value is based on the estimation of the CDF of the inter-arrival times using their cumulative frequencies.

A point that should be mentioned here is that a consecutively numbered message-id is appended to every heartbeat message to ensure that the heart-

beats are processed in the right order. If the monitoring process receives a heartbeat with a lower message-id than any id it previously received, then this obsolete message is just ignored. It makes no sense to set the current freshness point according to this outdated heartbeat regarding it as a new "life sign" from the monitored process. The use of message-ids to deal with message disorder is omitted in the specification of the failure detection algorithm but however assumed to be applied.

Algorithm 1 Basic failure detection algorithm

```
1:  Process q:
2:      send heartbeat message to p every Δ_i
3:
4:
5:  Process p:
6:
7:      f = −1                                          ▷ freshness point
8:      S = nil                                         ▷ S is initialised as empty list
9:      η                                               ▷ max size of S (e.g. 1000)
10:
11:     procedure RCV_HB(m_j, t)                        ▷ receiving heartbeat m_j at time t
12:         if f = −1 then
13:             f = t
14:         else
15:             t_Δ = t − f
16:             f = t
17:             append t_Δ to S
18:             if size of S > η then
19:                 remove head of S
20:             end if
21:         end if
22:     end procedure
23:
24:     function P_fail(t)                              ▷ returns suspicion value of q at time t
25:         t_Δ = t − f
26:         |S^{t_Δ}| = number elements in S that are lower or equal t_Δ
27:         |S| = number of elements in S
28:         return |S^{t_Δ}| / |S|
29:     end function
30:
```

In the following, extensions and variations of the introduced algorithm are presented.

2.5 Extensions and variations of the algorithm

2.5.1 Variation for partially synchronous systems

Now a variation of the basic algorithm is presented that mainly aims for proving that the modified algorithm implements a failure detector of $\Diamond P$ in partially synchronous systems [CT96, DLS88] with a crash failure model. Therefore, the completeness and accuracy properties have to be investigated. The partial synchrony used in the following evolves from the existence of an upper bound on communication, but this bound is not known a priori. For deeper insights of the used partial synchrony model it is referred to [DLS88]. The modified algorithm is shown in Algorithm 2.

The modification of the algorithm is based on the introduction of a new variable β that is initialised with 0. The value of β is added to every inter-arrival sample before inserted to the sampling window S. Additionally, β is incremented by 1 every time the failure detector experiences a failure probability of 1 for the monitored process but receives a newer heartbeat later on from this process.

It is now shown that the above specified failure detection algorithm implements an eventually perfect ($\Diamond P$) failure detector in partially synchronous systems. For this, the *strong completeness* and the *eventual strong accuracy* have to be shown to be satisfied.

Chandra et al. [CT96] prove that any given failure detector that satisfies weak completeness can be transformed into a failure detector that satisfies strong completeness. Furthermore they prove that, if the failure detector satisfies one accuracy property, the transformed also does. In this way it is sufficient to show that the algorithm implements weak completeness and eventual strong accuracy.

Without loss of generality it can be assumed that a process is suspected to have crashed if its failure probability is 1. The determination of a value that indicates the threshold after which a process is suspected to have failed is necessary for the following proofs. This is due to the classification of failure detectors assuming traditional algorithms which do suspect or do not suspect a process to have crashed.

Definition 1. A failure detector has the property *weak completeness* if eventually every process that crashes is permanently suspected by some correct process.

Theorem 1. *Assuming p is correct, q has crashed and p is monitoring q. Then the algorithm described in Algorithm 2 ensures that p eventually suspects q permanently.*

Algorithm 2 A failure detection algorithm of class $\Diamond P$

```
1:  Process q:
2:      send heartbeat message to p every Δ_i
3:
4:
5:  Process p:
6:
7:      f = −1                                          ▷ freshness point
8:      S = nil                                         ▷ S is initialised as empty list
9:      η                                               ▷ max size of S (e.g. 1000)
10:     β = 0          ▷ added to the inter-arrival times before inserted into S
11:
12:     procedure RCV_HB(m_j, t)                        ▷ receiving heartbeat m_j at time t
13:         if f = −1 then
14:             f = t
15:         else
16:             t_Δ = t − f
17:             t_Δ = t_Δ + β
18:             f = t
19:             append t_Δ to S
20:             if size of S > η then
21:                 remove head of S
22:             end if
23:         end if
24:     end procedure
25:
26:     function P_fail(t)                              ▷ returns suspicion value of q at time t
27:         t_Δ = t − f
28:         |S^{t_Δ}| = number elements in S that are lower or equal t_Δ
29:         |S| = number of elements in S
30:         return |S^{t_Δ}| / |S|
31:     end function
32:
33:     if suspicion value reaches 1 and further heartbeat is received then
34:         β = β + 1
35:     end if
36:
```

Proof. After q has crashed it stops to send heartbeat messages to p. Let t denote the time after the last heartbeat that has been sent by q to p has been received. Let t_{max} be the maximal inter-arrival time in the sampling window S. According to Algorithm 2 the suspicion value P_{fail} becomes 1 after $t \geq t_{max}$. Because of the partial synchrony the condition $t \geq t_{max}$ will

hold within a finite time after q's crash. Thus p will eventually suspect q. As q has crashed, p will not receive a further message from q. Therefore p will eventually suspect q permanently. □

Definition 2. A failure detector has the property *eventual strong accuracy* if there is a time after which correct processes are not suspected by any correct process.

Theorem 2. *Within a partially synchronous system the algorithm described in Algorithm 2 ensures that there is a time t after which no correct process is suspected to have crashed by a correct process.*

Proof. Let $p_1, \ldots, p_n$ be n processes where any two processes are monitoring each other. Let p_i and p_j be two random chosen correct processes where $i \neq j$ and p_i is monitoring p_j. With every false suspicion of p_j, the process p_i increments β. Therefore, p_i cannot suspect p_j infinitely often. As p_i only may suspect p_j finitely often and p_j is a correct process, eventually p_i will receive a further heartbeat and stop to suspect p_j permanently. □

Theorem 1 and Theorem 2 show that the modified algorithm implements a failure detector of class $\Diamond P$. The modifications of the algorithm that has been proposed in this paragraph cause a better accuracy at the expense of a longer failure detection time. This modification is particularly useful for systems where it must be ensured that the failure detector eventually makes no false detections. For the unmodified basic algorithm, only the strong completeness property holds.

2.5.2 Self-adjusting failure detector

The variation of the failure detector proposed in this section reconsiders previously computed failure prognoses based on the actual events. This information is used to adjust the failure detector online, in order to deliver better results.

More precisely, it records the average suspicion value $out_{\varnothing}$ of all requests it answered so far together with the total number of requests out_{num}. Furthermore it counts the number of all errors out_{err} it made, i.e. all occasions where it outputted a suspicion value greater than 0 and recognised later on that the monitored service did not fail. Whenever $out_{\varnothing} > 1 - \frac{out_{err}}{out_{num}}$ holds, β is incremented by a small value σ, e.g. $\sigma = \frac{\Delta_i}{10^4}$ and $out_{\varnothing}$ is reset to 0.

This modification of the basic algorithm, as shown in Algorithm 3, avoids an overestimation of failure probabilities at the cost of a fast failure detection. Overestimated suspicions are often considered to be the bigger problem in real systems, where an overestimation could result in costly repair

mechanisms which might also be difficult to undo later on. Furthermore it is well-suited for environments where failures and recoveries of nodes occur frequently which enables a better evaluation of the correctness of the computed suspicion values.

Algorithm 3 Self-adjusting failure detector

1: **Process** q:
2: send heartbeat message to p every Δ_i
3:
4: **Process** p:
5: $f = -1$ ▷ freshness point
6: $S = nil$ ▷ S is initialised as empty list
7: η ▷ max size of S (e.g. 1000)
8: $\beta = 0$ ▷ added to inter-arrival times before inserted into S
9: $\sigma = \frac{\Delta_i}{10^4}$ ▷ the incrementation factor of β
10: $out_{\varnothing}$ ▷ average of computed suspicion values
11: out_{err} ▷ number of errors
12: out_{num} ▷ number of outputted suspicion values
13:
14: **procedure** RCV_HB(m_j, t) ▷ receiving heartbeat m_j at time t
15: **if** $f = -1$ **then**
16: $f = t$
17: **else**
18: $t_\Delta = t - f$
19: $t_\Delta = t_\Delta + \beta$
20: $f = t$
21: append t_Δ to S
22: **if** size of $S > \eta$ **then**
23: remove head of S
24: **end if**
25: **end if**
26: **if** $out_{\varnothing} > 1 - \frac{out_{err}}{out_{num}}$ **then**
27: $\beta = \beta + inc$
28: $out_{\varnothing} = 0$
29: **end if**
30: **end procedure**
31:
32: **function** $P_{fail}(t)$ ▷ returns suspicion value of q at time t
33: $t_\Delta = t - f$
34: $|S^{t_\Delta}|$ = number elements in S that are lower or equal t_Δ
35: $|S|$ = number of elements in S
36: $out_{num} = out_{num} + 1$
37: $out_{\varnothing}$ = average of $\frac{|S^{t_\Delta}|}{|S|}$ since last error
38: **return** $\frac{|S^{t_\Delta}|}{|S|}$
39: **end function**
40:

2.5.3 Freshness point strategy

The variation of the basic failure detection algorithm presented in the following abolishes the dependence of the suspicion values on the arrival times of past heartbeats. This negative property of many failure detectors is addressed by Chen et al. [CTA00]. The techniques of heartbeat sampling and the freshness point strategy, as used in Algorithm 1, entail the dependence of the failure probability on the previous heartbeat. Assuming again p is monitoring q and p is waiting for the i-th heartbeat from q, then the failure probability of q not only depends on the arrival time of the i-th heartbeat m_i, but also on the past receipt time of the $(i-1)$-th heartbeat m_{i-1}. In fact, this time has a big influence on the current failure probability. If m_{i-1} has arrived "fast" then p is waiting a longer time for m_i since the last freshness point which is the receipt time of m_{i-1} has been set early. Hence, the failure probability will become higher as if m_{i-1} had arrived "late". In the latter case the freshness point is set later and therefore the failure probability is lower.

To circumvent this dependency it is proposed to use a different concept of freshness points and inter-arrival times. Freshness points should not be set according to the arrival time of heartbeat messages, which is subject to network variation, to avoid the above mentioned drawbacks. A better choice is to use the sending time according to the sender's local clock instead. Then, however, the inter-arrival times have to be computed differently, too. Let m_i denote the arrival times of heartbeats according to p's local clock and s_i its corresponding sending times according to q's clock. The inter-arrival times used in the basic algorithm can be computed as $m_i - m_{i-1}$. Inter-arrival times appropriate for the new freshness point strategy are calculated as $m_i - s_{i-1}$.

Table 2.2 illustrates an example of four heartbeat messages with corresponding sending times s_i, receiving times m_i, actual freshness point f, and sampling window S according to the common freshness point strategy. Table 2.3 represents the same example using the new strategy. The values of S in Table 2.3 are generally higher because the sending times are included in the inter-arrival times. But this has no negative influence on the failure detection process as only the variations of these values are crucial. In the first table it is noticeable that the previous heartbeat greatly influences the inter-arrival times. For example, the fourth heartbeat arrives rather early. In Table 2.3 this results in a rather small inter-arrival time of 103. In Table 2.2, however, this resulted in a comparatively much smaller value because the fourth heartbeat arrives late.

Please note that the method introduced here to abolish the dependence on the last heartbeat is not based on synchronised clocks.

i	s_i	m_i	f	S
1	1000	1010	1010	[]
2	1100	1105	1105	[95]
3	1200	1230	1230	[95, 125]
4	1300	1303	1303	[95, 125, 73]

Table 2.2: *Common freshness point strategy*

i	s_i	m_i	f	S
1	1000	1010	1000	[]
2	1100	1105	1100	[105]
3	1200	1230	1200	[105, 130]
4	1300	1303	1300	[105, 130, 103]

Table 2.3: *New freshness point strategy*

Algorithm 4 shows the basic algorithm presented in Algorithm 1 modified according to the concepts presented here.

The presented mechanism to abolish the dependence on the last heartbeat works for all failure detectors that are based on sampling heartbeat inter-arrival times sent from the monitored to the monitoring process. The evaluation in Section 2.6 shows that this variation improves the failure detector. It can be assumed that this positive effect also holds for other algorithms that sample inter-arrival times like the φ failure detector [HDYK04].

2.5.4 Sampling window

In this section, variations based on a different representation, interpretation, and manipulation of the sampled inter-arrival times are discussed.

Histogram bin width

In the basic algorithm, as described in Algorithm 1, the sampling window is a list containing the last η sampled inter-arrival times. To compute a failure probability, the cumulative frequencies of the entries in the sampling window are used. The resolution level of the cumulative frequencies is at the resolution of the data - no certain bin width is used to cluster the data. The procedure of the computation of a failure probability is illustrated in Figure 2.7. In this example the probability of a failure is calculated for a failure detector whose current freshness point is two seconds old.

In the basic version of the failure detection algorithm no histogram in the actual sense is used. The sampling window is just a list containing the

Algorithm 4 Failure detection algorithm with a different freshness point strategy

1: **Process** q:
2: send heartbeat message to p every Δ_i
3:
4:
5: **Process** p:
6:
7: $f = -1$ ▷ freshness point
8: $S = nil$ ▷ S is initialised as empty list
9: η ▷ max size of S (e.g. 1000)
10: t_s ▷ the sending time according to q's clock
11:
12: **procedure** RCV_HB(m_j, t) ▷ receiving heartbeat m_j at time t
13: **if** $f = -1$ **then**
14: $f = t_s$
15: **else**
16: $t_\Delta = t - f$
17: $f = t_s$
18: append t_Δ to S
19: **if** size of $S > \eta$ **then**
20: remove head of S
21: **end if**
22: **end if**
23: **end procedure**
24:
25: **function** $P_{fail}(t)$ ▷ returns suspicion value of q at time t
26: $t_\Delta = t - f$
27: $|S^{t_\Delta}|$ = number elements in S that are lower or equal t_Δ
28: $|S|$ = number of elements in S
29: **return** $\frac{|S^{t_\Delta}|}{|S|}$
30: **end function**
31:

inter-arrival times; the cumulative frequencies are used to calculate a failure probability, the resolution of this calculation is at the level of the sampled data. But there are some advantages that come along with the division of the data in bins. Then, the sampling window does not consist of the values of the sampled data, but only the information how many values a bin contains. For instance $S = [1.083s, 0.968s, 1.062s, 0.993s, 0.942s, 2.037s, 0.872s]$ could become to $S = [0s, 1s) : 4$, $[1s, 2s) : 2$, $[2s, 3s) : 1$, while $[[0s, 1s) : 4)$ denotes four arrivals in the time interval $[0s, 1s)$. It is obvious that the use of such a representation of the data allows for a faster generation of a failure

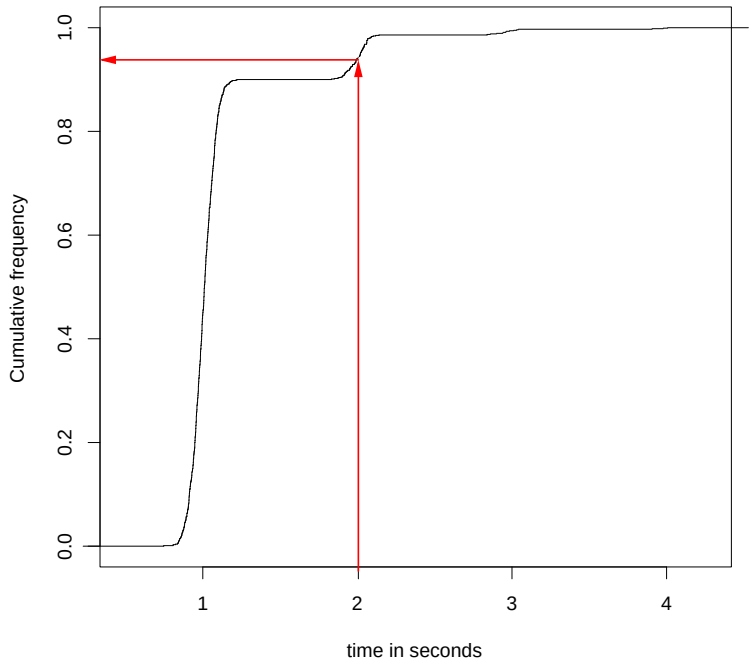


Figure 2.7: *Computation of a failure probability*

probability. That is valid due to the fact that the counting of elements that are lower or equal to a certain elapsed time only depends on the number of bins which is typically clearly smaller than the number of samples.

Another advantage does only comes into effect if the sampling window's size is not limited: the coarsening of the granularity of the resolution of the data saves a big amount of memory. If, however, the sampling window has a specific size, values that are inserted have to be deleted later on when they become too old. Thus every data value has to be saved explicitly and the memory-saving is invalid. The drawback of dividing samples into bins is the loss of information that might cause the failure detector to perform worse, particularly if the bin width is chosen too big.

Figure 2.8 shows four examples of a sampling window as histograms with different bin widths. Figure 2.9 illustrates the effects of the different bin widths on the cumulative frequencies which are used to compute the suspicion values.

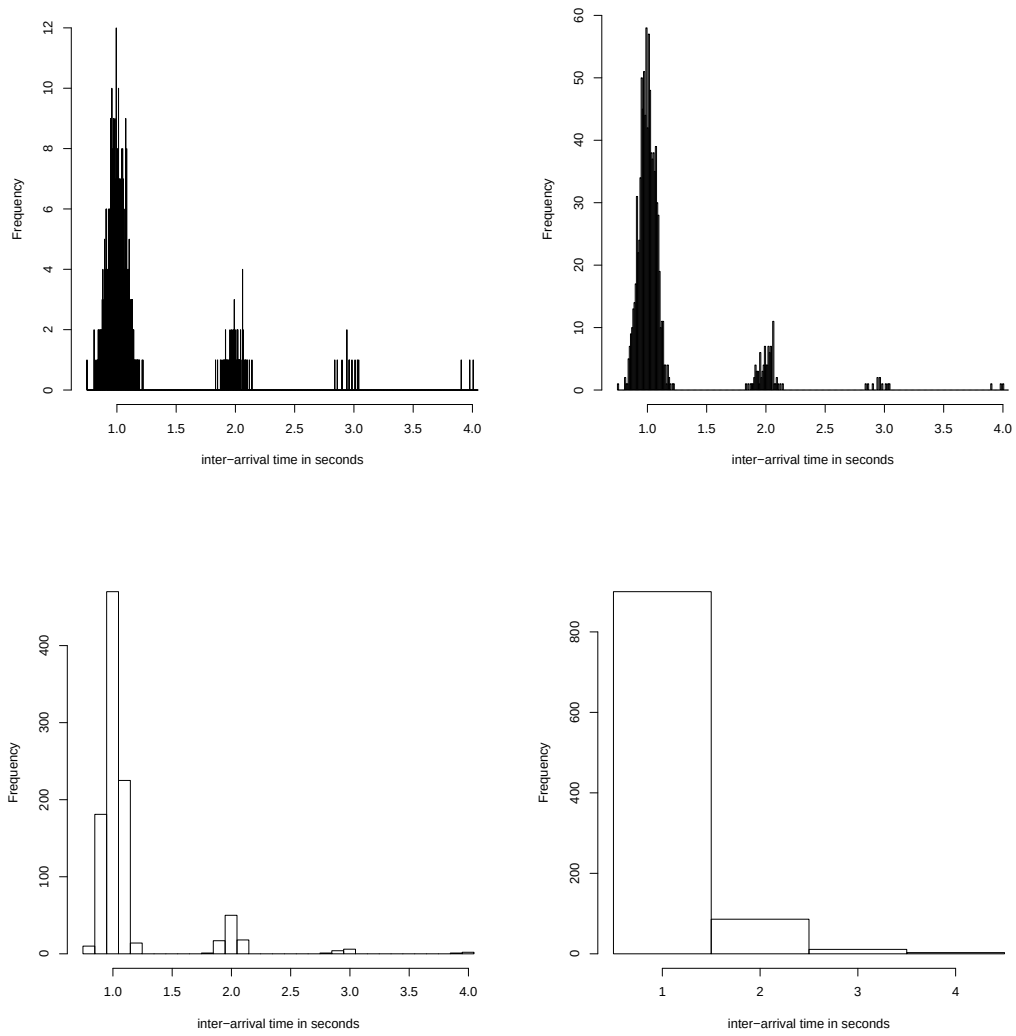


Figure 2.8: *Sampling window as histograms with different bin widths*

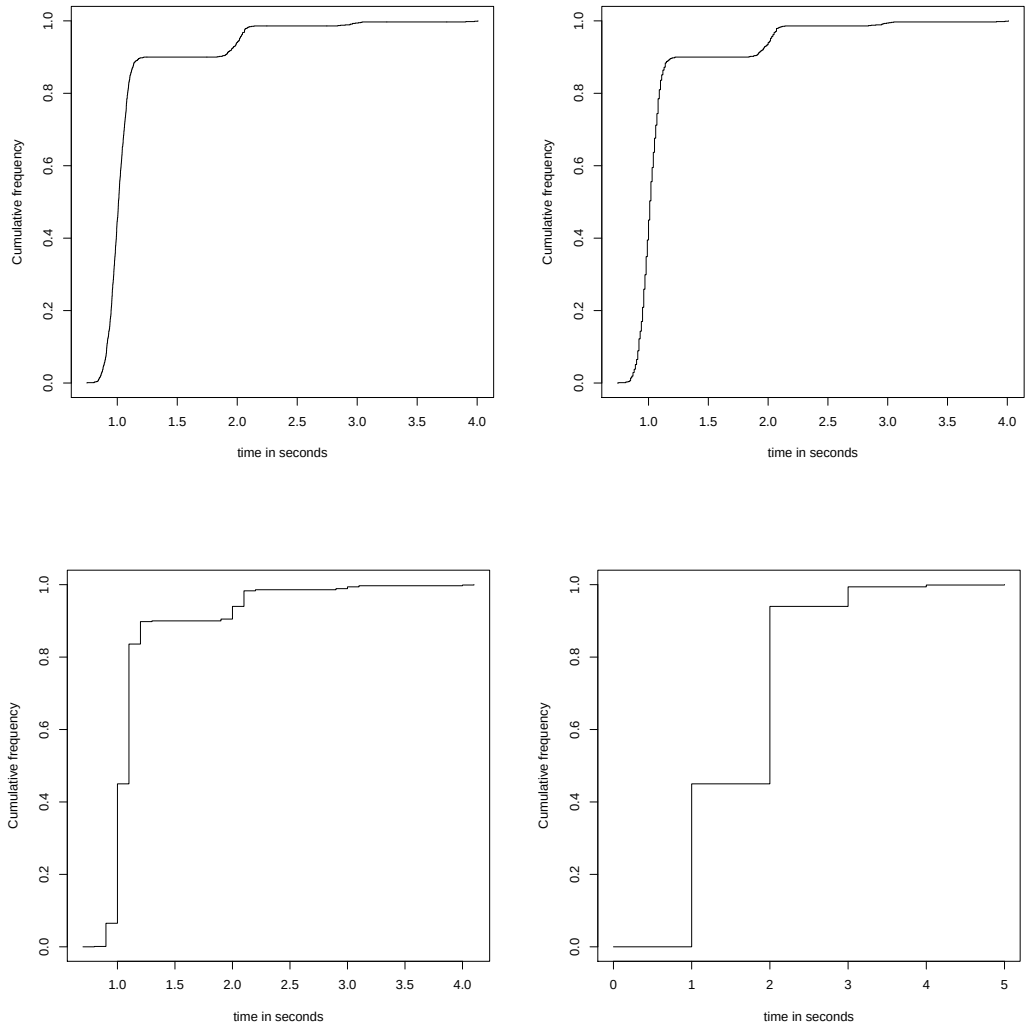


Figure 2.9: *Cumulative frequencies based on histograms with different bin widths*

Histogram smoothing

The heartbeat data sampled over time are subject to random variations due to for instance the unpredictable behaviour of the network. There are methods for reducing the effects of random variations on sampled data. The purpose for this is to reveal more clearly the underlying basic distribution. A technique that can be used to achieve this is called *smoothing* and can be used to smooth histograms.

A smoother can be seen as a kind of a weighted averaging process. The aiming value is transformed by an averaging of the values in its neighbourhood. The size of the neighbourhood that is taken into account has to be set in an appropriate way. The parameter characterising this amount of neighbouring values is called *smoothing parameter*. Generally, the larger the smoothing parameter, the smoother the result.

A simple yet fast smoother has been chosen to be used together with the failure detector and is described in the following. Each band of the histogram is smoothed by averaging over a moving window. The smoothing parameter k determines the size of the moving window which is set to $2k + 1$. If the window runs off the end of the histogram, bands of size 0 are considered. The process of this smoothing technique is illustrated in Figure 2.10. The blue area represents the sampling window - only values within this window are taken into account. The aiming value is the value in the centre of the sliding window and highlighted in red. This centre value is now set to the average of all values within the sliding window. The sliding window moves through the hole histogram until all bands are processed.

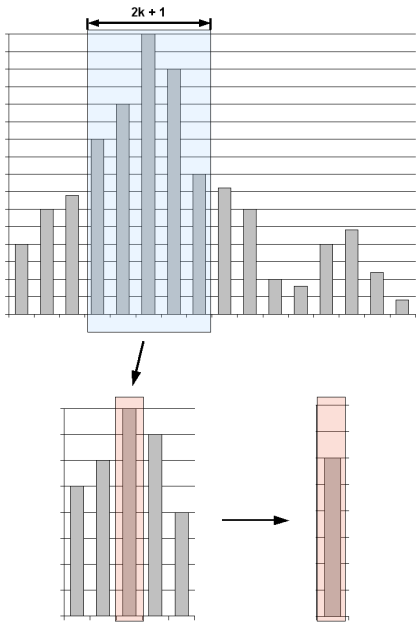
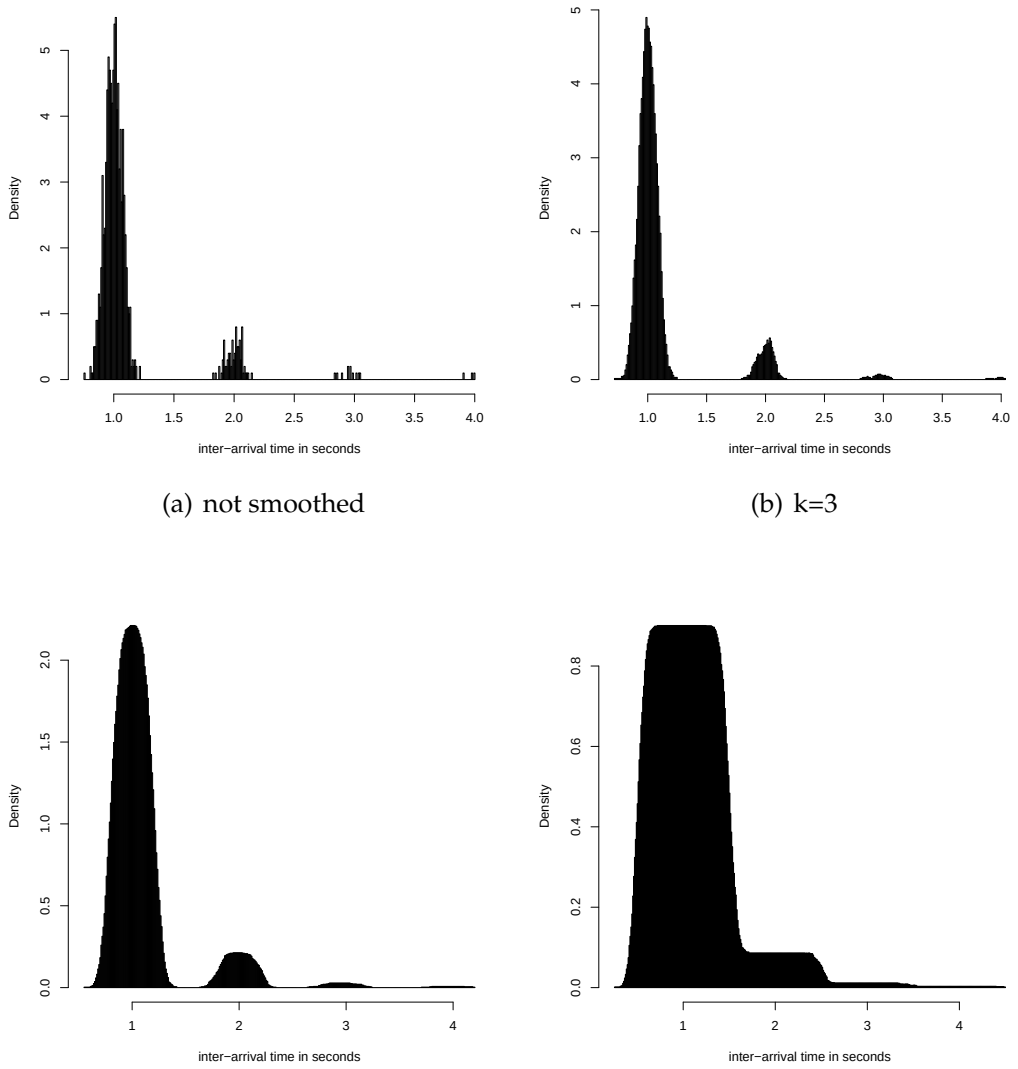


Figure 2.10: *Histogram smoothing*



(a) not smoothed

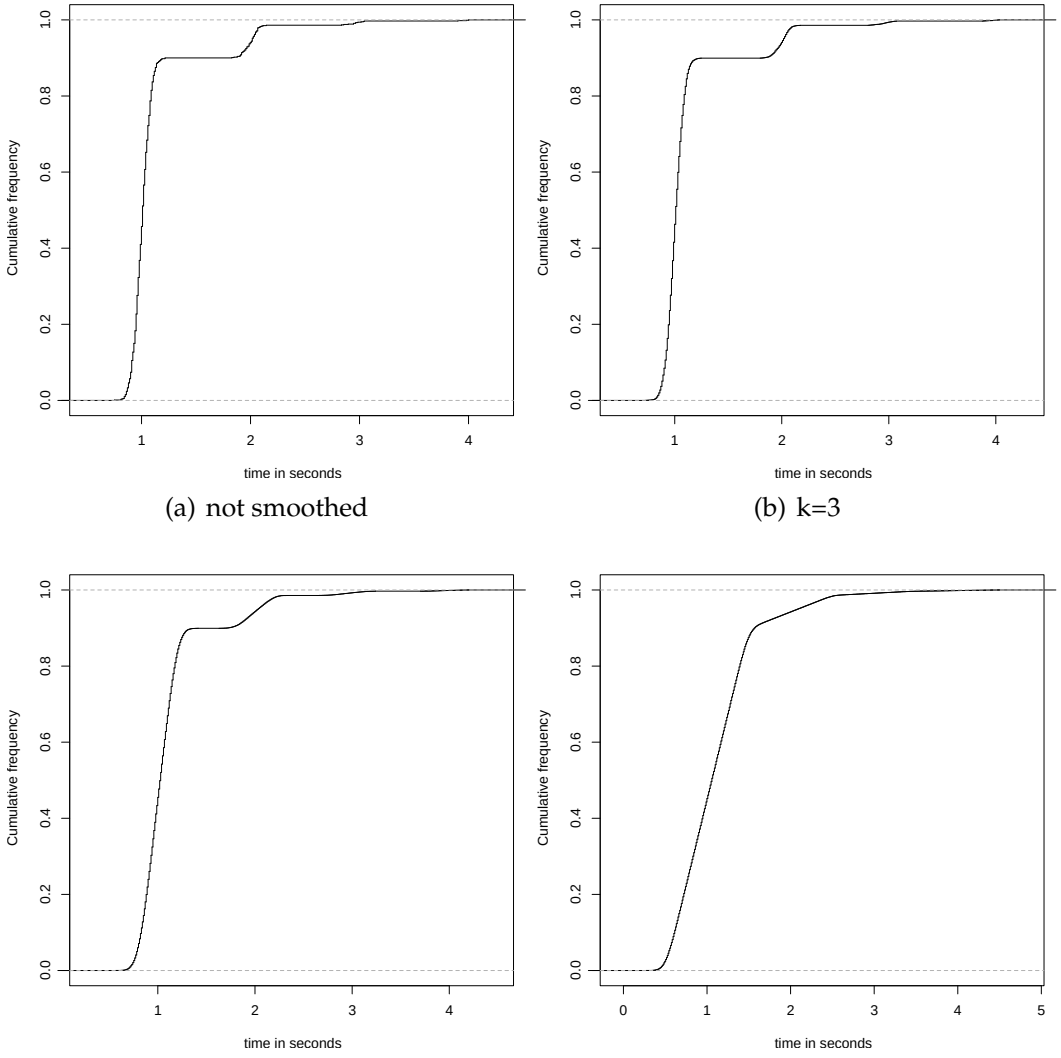
(b) k=3

(c) k=20

(d) k=50

Figure 2.11: *Smoothed histograms*

For further readings about smoothing techniques it is referred to [Här91, Sil86]. In Figure 2.11, histograms smoothed with different smoothing parameters are illustrated. Figure 2.12 shows the corresponding cumulative frequencies. Obviously, the choice of the smoothing parameter is important. The larger the value k, the smoother the resulting histogram. However, if k is chosen too large over-smoothing occurs with loss of essential histogram features. The usage of histogram smoothing only changes the failure detection algorithm in calculating the failure probability. It is based on the cumulative frequencies of the smoothed histogram instead of the unsmoothed cumulative frequencies.



(a) not smoothed (b) k=3

(c) k=20 (d) k=50

Figure 2.12: *Cumulative frequencies based on smoothed histograms*

Using a smoothing technique can cause the failure detector to be more robust to random variations. Also a larger sampling window results in the failure detector being less influenced by temporary extreme conditions. On the other hand real changes of the conditions of the systems are met slower if the sampling window is large because values that have been experienced in the past have still a huge influence on the results of the failure detector.

In the next section an evaluation for the proposed failure detection algorithms is provided.

2.6 Evaluation

In this section a methodology is introduced to evaluate failure detectors. Based on these concepts, a *test centre* for heartbeat-style failure detection has been developed. Using this tool, a series of performance measurements have been conducted to compare the developed failure detector against other state of the art failure detectors and to measure the effects of the proposed variations.

The basic methodology for investigating the value of a failure detector for a certain domain contains the following two steps:

Qualitative Evaluation: In the first step a qualitative approach is used to examine fundamental properties of failure detectors and their suitability for a certain application area.

Quantitative Evaluation: Within a quantitative evaluation process the performance of failure detectors applied to a certain domain is measured. To conduct such measurements, the test centre can be used.

In the following evaluation, the basic failure detection algorithm presented in this work (see Algorithm 1) is compared with the well known failure detectors of Chen et al. [CTA00] and Bertier [BMS02], and the accrual φ failure detector of Hayashibara et al. [HDYK04]. All these algorithms use a heartbeat-style approach to monitor processes.

2.6.1 Qualitative Evaluation

For the qualitative evaluation, the following categories are considered:

Adaptive: Adaptive failure detectors are able to adapt to changing network conditions (see Section 2.3.4).

Network load: It is examined to what extent failure detectors are able to influence the network load they are producing.

Accrual: Accrual failure detectors output a suspicion information on a certain scale rather than outputting a binary value. This results in more flexible failure detectors. (see Section 2.3.6)

Computational complexity: The computational complexity of a failure detector is an important feature. Both the complexity computing a suspicion value on enquiry of the monitoring process as well as the processing of a received heartbeat message has to be taken into account.

Table 2.4 shows the results of the qualitative evaluation according to the above categories, where η is the maximal size of the sampling window S.

Algorithm	Adaptive	Network load	Accrual	Computational complexity	
				query	heartbeat
Basic	✔	tailorable & lazy	✔	$\mathcal{O}(\eta)$ $\mathcal{O}(log\,\eta)$	$\mathcal{O}(1)$ $\mathcal{O}(log\,\eta)$
Chen	✔	tailorable	✘	$\mathcal{O}(\eta)$	$\mathcal{O}(1)$
φ	✔	tailorable	✔	see text	$\mathcal{O}(1)$
Bertier	✔	tailorable	✘	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Table 2.4: *Qualitative analysis*

All algorithms can be called adaptive because they all have the ability to adapt to changing network conditions. Additionally, all failure detectors are tailorable as they all can downgrade their detection quality in order to reduce the network load. The algorithms are based on periodically sent heartbeat messages. An adjustment of this heartbeat interval is the instrument to affect the network load in this way.

The only failure detector however, that comes with an approach for lazy monitoring is the failure detector presented in this work. The lazy monitoring is described in Section 2.7 and enables the failure detector to reduce the network load without the negative effects on the detection quality.

Accrual failure detectors output a suspicion value in place of a binary value. They are more flexible and can be seen as a generalisation of conventional failure detectors. Accrual failure detectors can easily be transformed into conventional binary failure detectors by defining a threshold. Values below this threshold imply *not-failed* and values above *failed* as output. The failure detector presented in this work and the φ failure detectors belong to the class of accrual failure detectors while the other two algorithms are conventional failure detectors.

For all algorithms the complexity of processing a heartbeat belongs to $\mathcal{O}(1)$. The complexity of answering a query, i.e. computing a suspicion information for Bertier's algorithm is $\mathcal{O}(1)$ and for Chen's it is $\mathcal{O}(\eta)$. Basically, the complexity of answering a query for the basic algorithm is also $\mathcal{O}(\eta)$, but with the usage of for example a sorted list to implement the sampling window it is $\mathcal{O}(log\,\eta)$. However, the usage of such a data structure extends the complexity for a heartbeat arrival also to $\mathcal{O}(log\,\eta)$. Thus, if the failure detector is often queried and the heartbeat interval is long then the use of e.g. a sorted list can provide a benefit, otherwise the use of a non-sorted data structure with an inexpensive insert operation like a standard list should be preferred. The complexities do not change for the presented variations of the algorithm except the variations that partition the data into bins and the application of a smoothing technique. Assuming the sampling window

is partitioned into η_b bins, then the computation of a suspicion value takes $\mathcal{O}(\eta_b)$, the processing of a heartbeat remains in $\mathcal{O}(1)$. Using a smoothing technique the complexity of this technique has to be taken into account additionally.

The computation of a suspicion value of the φ failure detector of Hayashibara et al. [HDYK04] includes the following steps:

1. Determining the mean μ of the sampled values
2. Determining the variance σ^2 of the sampled values
3. Computation of
$$P_{later}(t) = \frac{1}{\sigma\sqrt{2\pi}} \int_t^{+\infty} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx$$
4. Computation of $\varphi(t_{now}) = -log_{10}(P_{later}(t_{now} - T_{last}))$

Thus the computational complexity of the φ failure detector depends on the used methods to compute the above listed steps. Step 3 should be the determining factor for the computational complexity if the mean and variance of the sampled values are computed recursively. For sampling windows of a common moderate size it could be assumed that the computation of a suspicion value according to the φ failure detector takes significantly longer than according to the other failure detectors presented in Table 2.4.

Now a quantitative evaluation follows, measuring the performance of failure detection algorithms.

2.6.2 Quantitative Evaluation

At first, the test centre for failure detectors developed to measure the performance of heartbeat-style failure detection algorithms is described. Afterwards, the results of the performance measurements are presented.

Test centre

The test centre is a tool that allows for an easy evaluation of failure detection algorithms that are based on heartbeat messages. Figure 2.13 gives an overview of the different functional parts of this framework.

In a nutshell the evaluation centre takes as input heartbeat samples and applies a failure detection algorithm to it. As result the performance of the failure detector according to a set of metrics is outputted.

In the following the functional parts of the test centre are described:

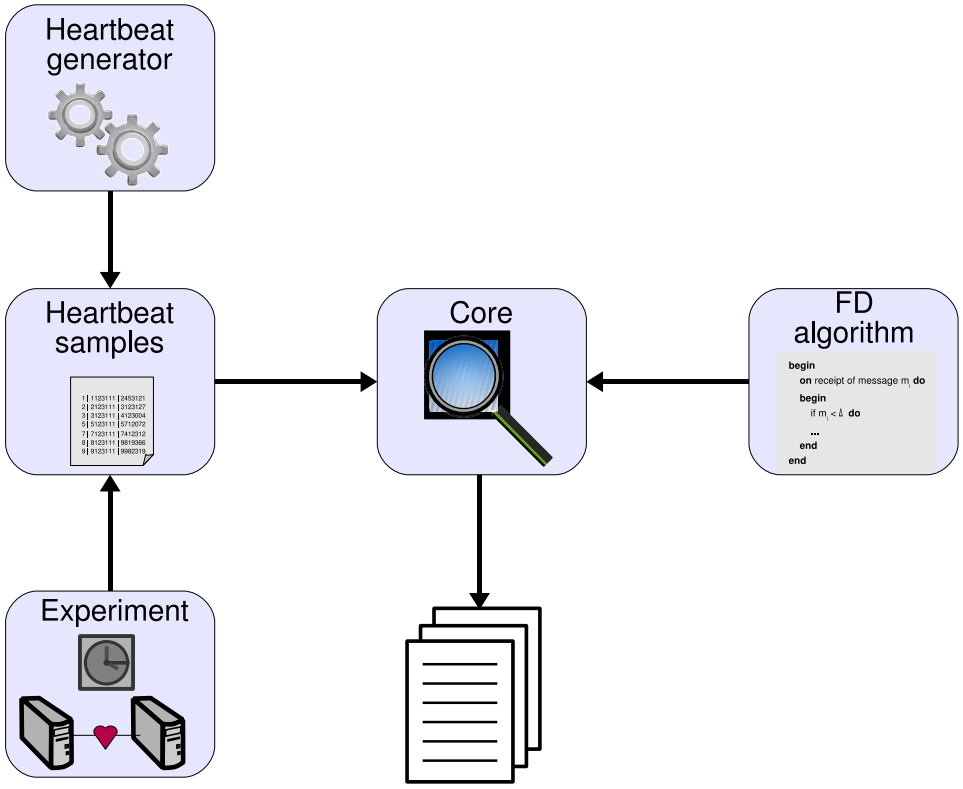


Figure 2.13: *Test centre for failure detection algorithms*

Heartbeat samples Again it is assumed that p and q are processes where p is monitoring q. The heartbeat samples consist of data that give information about the heartbeat sending process. In more detail it is a dataset consisting of entries in the format:

```
heartbeatid | sendingtime | arrivaltime
```

`heartbeatid` is a consecutive id which q appends to every heartbeat to enable p to detect lost and late heartbeats. `sendingtime` is the time the heartbeat has been sent according to q's local clock. `arrivaltime` is the time the heartbeat has been received according to p's local clock. If a heartbeat gets lost then `arrivaltime` is empty.

The heartbeat samples have a very simple format but contain all relevant information about the environment like the message delay and loss of the network as well as q's ability to send heartbeats at the right time.

Heartbeat samples can be produced either by a setup in a real environment (experiment) or by generating these data artificially (heartbeat generator).

Experiment The best way to evaluate how failure detectors perform in a certain environment is to produce the heartbeat samples within this environment. To do so, the heartbeat sending and receiving times of two processes have to be logged and provided in the above stated format.

Heartbeat generator An infinite set of environments exists, regarding the computing devices and their interconnection which influence the performance of failure detection algorithms. For a more generic evaluation, the heartbeat samples needed for the evaluation can be generated. This has the benefit that the experiments are reproducible and independent of any special properties of e.g. a certain type of communication medium.

Formula 2.2 describes how the sending time of the j-th heartbeat t_s^j is computed. The heartbeat interval is denoted with Δ_i, ϱ stands for a random variable sampled according to a certain probability distribution to model the lag of time of q sending heartbeats. The test centre provides the possibility to sample according to a set of probability distributions, such as the Normal, Log-Normal, Exponential, Gamma, and Weibull distribution.

$$t_s^j = t_s^{j-1} + \Delta_i + \varrho \tag{2.2}$$

The generation of the arrival time of the j-th heartbeat t_r^j is described with Formula 2.3. To model the sending delays of the heartbeat messages again a random variable δ is used. θ represents the loss probability of a heartbeat. For the message loss probability two different approaches are available: conditional and unconditional message loss [Bol93]. In the case of a conditional message loss model, the probability of a message loss changes depending whether the previous message is lost or not. This may be used to model bursty message loss behaviour. Unconditional message loss does not make this subdivision, all messages are subject to a message loss with the same probability regardless of previous events. The heartbeat generator of the test centre uses a conditional message loss model as this is a more generic approach that covers the unconditional loss model as one special case.

$$t_r^j = \begin{cases} t_s^j + \delta & , \quad probability : 1 - \theta \\ message\ loss & , \quad probability : \theta \end{cases} \tag{2.3}$$

Formula 2.4 shows the computation of the message loss probability. To specify a message loss behaviour, the overall message loss probability χ and a

burstiness factor κ have to be declared. The probability of a message loss under the assumption that the previous message has been lost is $\kappa \cdot \chi$. If the previous message has not been lost then the failure probability is $\frac{\chi - \kappa \cdot \chi^2}{1-\chi}$. Thus the overall failure probability is χ, according to the total probability theorem [Pap84].

$$\theta = \begin{cases} \kappa \cdot \chi & , \quad \textit{previous message lost} \\ \frac{\chi - \kappa \cdot \chi^2}{1-\chi} & , \quad \textit{previous message not lost} \end{cases} \tag{2.4}$$

κ set to 1 results in an unconditional message loss behaviour. In this case the probability of a message loss is not influenced by the previous message:

$$\theta = \begin{cases} \kappa \cdot \chi = \chi & , \quad \textit{previous message lost} \\ \frac{\chi - \kappa \cdot \chi^2}{1-\chi} = \frac{\chi - \chi^2}{1-\chi} = \frac{\chi \cdot (1-\chi)}{1-\chi} = \chi & , \quad \textit{previous message not lost} \end{cases}$$

Failure detection algorithm To evaluate a failure detection algorithm, an implementation of this algorithm has to be provided to the test centre. If a new failure detector is evaluated this is the only part where programming is necessary. To minimise this work, an abstract failure detection class is provided in which solely two methods need to be overwritten.

Core The test centre applies the specified failure detection algorithm to certain heartbeat samples measuring its performance according to a set of metrics. In order to compare accrual and non-accrual failure detectors, the accrual failure detectors have to be transformed into conventional failure detectors. Therefore, a threshold T is specified. If the level of suspicion of an accrual failure detector is lower than this threshold, then the process is not suspected to have failed. If the level of suspicion crosses T, it is suspected.

The next barrier to compare failure detection algorithms are their different tuning parameters. These influence the time when a failure detector starts/ends to suspect a process. For the accrual failure detectors the tuning parameter is in this case the threshold T. Non-accrual failure detectors often have explicit tuning parameters. Chen's failure detector for example has as tuning parameter a safety margin α. This is a constant period of time that is added to the estimated heartbeat arrival time. Other failure detectors like Bertier's have no tuning parameters. To be able to compare the different failure detection algorithms, the behaviour of the failure detectors using several values of their respective tuning parameters should be measured.

To compute metrics concerning the detection time of the failure detectors it is assumed that a crash would occur always exactly after successfully send-

ing a heartbeat message. Then the time it takes until the failure detector reports a suspicion is measured. This corresponds to the worst case situation. This method to compute the worst-case detection time is also used in [HDYK04]. To evaluate metrics concerning the number of mistakes exactly the same experiment is considered but under the assumption that no single crash occurs.

Evaluation results The output of an evaluation run of the test centre is the performance of the failure detection algorithm according to the following metrics implemented as introduced in Section 2.3.2:

- Number of mistakes N_M,
- average detection time T_D,
- average mistake recurrence time T_{MR},
- average mistake duration T_M,
- average mistake rate per second λ_M,
- query accuracy probability P_A, and
- average good period duration T_G.

An example of such an output for the basic failure detector, transformed into a non-accrual failure detector with threshold $T = 0.97$, is shown in Figure 2.14.

```
- Basic FD - (threshold: 0.97)
num mistakes: 9315
avg detection time: 20002.62696640515
avg mistake recurrence time: 1067437.1914322525
avg mistake duration: 10642.299409554482
avg mistake rate per second: 9.36708097019428E-10
query accuracy probability: 0.9900312719721652
avg good period duration: 1056770.4754696726
```

Figure 2.14: *Exemplified output of the test centre*

The test centre is implemented in JAVA. Figure 2.15 shows a class diagram of interesting parts of the test centre. Not previously mentioned is the class `EvaluationPlan` that allows for the specification of plans for the test centre that are sequentially executed as single evaluation runs in a kind of batch mode.

In order to be evaluable by the test centre, every failure detection algorithm has to implement the abstract class `AbstractFD`. Therefore, the two methods `newMessage` and `getNextFreshnessPoint` must be implemented.

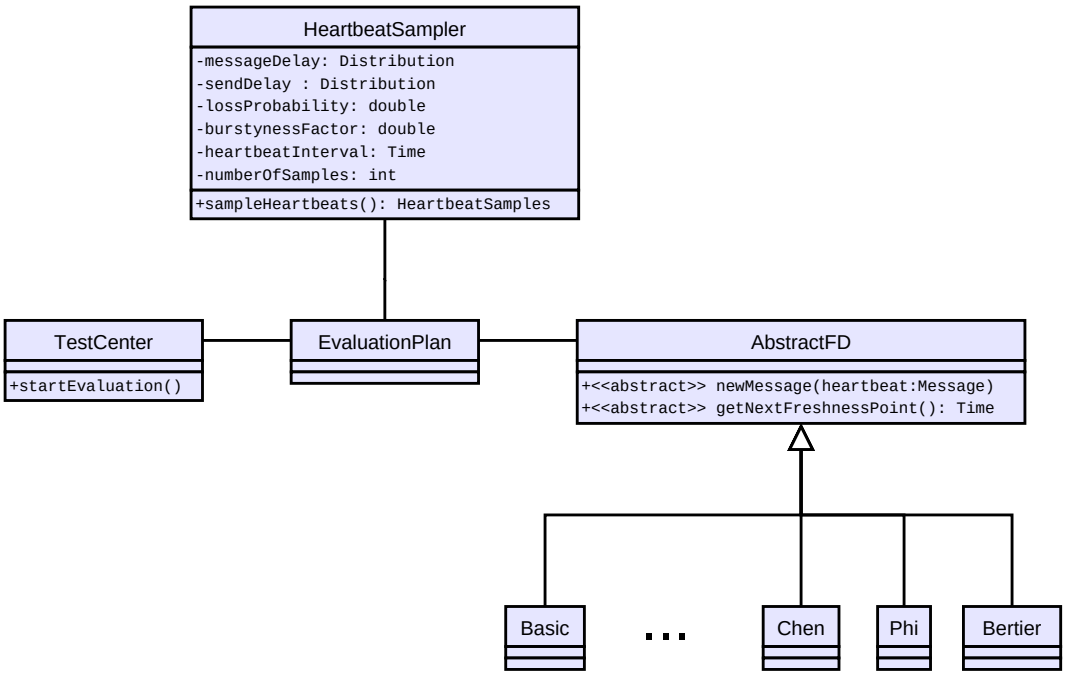


Figure 2.15: *Simplified class diagram of the test centre*

The method `newMessage` simulates the arrival of a new heartbeat while `getNextFreshnessPoint` must return the time when the failure detector starts to suspect the monitored process if no further message arrives. For many failure detectors these two methods can be implemented with a few lines of code. Besides this, the test centre provides all the functionality necessary to evaluate the failure detector's performance.

To perform one run of the test centre with the usage of the heartbeat sampler, the value for the heartbeat interval Δ_i and the size of the sampling window η has to be set.

If the heartbeat samples are generated with the use of the heartbeat sampler, additionally the following input parameters have to be specified:

- Number of heartbeats,
- message delay distribution δ,
- send delay distribution ϱ,
- message loss probability χ, and
- burstiness factor κ.

In the following the results of the performance measurements that have been conducted with the described test centre are presented.

Evaluation setting

In order to evaluate and compare the failure detection algorithms a scenario is chosen according to results of researchers investigating the messaging behaviour of the Internet.

Bolot [Bol93] and Mukherjee [Muk92] reason that the Internet end-to-end delay distribution they experienced in their experiments is best modelled by a shifted gamma distribution. Sanghi et al. [SAGJ93] encountered packet loss rates between 2.1% and 10.1% in their measurements. Dam et al. [DN98] selected a site in the US, sent ping packets at regular intervals and noted the RTT for each ping packet. The closest gamma distribution fit for the packet delay of this experiment turned out to be a shifted gamma distribution with shape parameter 2.0 and scale parameter 2.8, as shown in Figure 2.16.

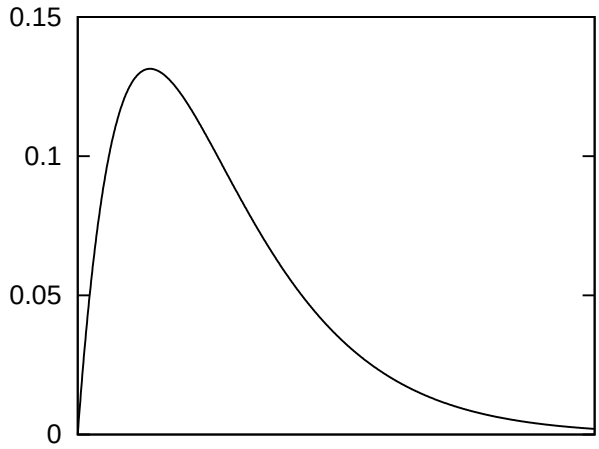


Figure 2.16: *Gamma distribution*

Basic algorithms vs. other state of the art algorithms In the following, 12 experiments are presented to compare the basic failure detection algorithm against other state of the art failure detectors. Per experiment one million heartbeat messages are generated using a shifted gamma distribution with shape parameter 2.0 and scale parameter 2.8 to model the message delay. The heartbeat interval Δ_i is set to 10 seconds in all experiments. These differ in the modelling of the message loss and the size of the sampling window η.

In Experiment 1.1 - 1.6 the sample window size η is set to 1000 samples for all algorithms. This means that the computations of the failure detectors rely only on the last 1000 heartbeat message samples. Furthermore, the

measurements are started not until 1000 heartbeats have been received to grant a warm-up phase.

The first three experiments examine the performance of the failure detection algorithms with different message loss probabilities but unconditional non-bursty message loss behaviour.

Experiment 1.1: η: 1000, χ: 2%, κ: 1	(Figure 2.17)
Experiment 1.2: η: 1000, χ: 5%, κ: 1	(Figure 2.18)
Experiment 1.3: η: 1000, χ: 10%, κ: 1	(Figure 2.19)

The following three experiments consider a conditional bursty message loss with a burstiness factor of $\kappa = 5$. Thus message loss in the case of a previously lost message has the probability $5 \cdot \chi$.

Experiment 1.4: η: 1000, χ: 2%, κ: 5	(Figure 2.20)
Experiment 1.5: η: 1000, χ: 5%, κ: 5	(Figure 2.21)
Experiment 1.6: η: 1000, χ: 10%, κ: 5	(Figure 2.22)

Experiments 1.7 - 1.12 are similar to the first six experiments, but the sample window size η is set to 20000 samples. Thus the effects of the sampling window size on the detection quality can be investigated. A warm-up phase of 20000 heartbeats is granted to all algorithms.

Experiment 1.7: η: 20000, χ: 2%, κ: 1	(Figure 2.23)
Experiment 1.8: η: 20000, χ: 5%, κ: 1	(Figure 2.24)
Experiment 1.9: η: 20000, χ: 10%, κ: 1	(Figure 2.25)
Experiment 1.10: η: 20000, χ: 2%, κ: 5	(Figure 2.26)
Experiment 1.11: η: 20000, χ: 5%, κ: 5	(Figure 2.27)
Experiment 1.12: η: 20000, χ: 10%, κ: 5	(Figure 2.28)

Within these settings the basic failure detector as described in Algorithm 1 is compared with the well known failure detectors of Chen et al. [CTA00] and Bertier [BMS02] and the accrual φ failure detector of Hayashibara et al. [HDYK04].

The test centre outputs for every run a file containing the results according to the metrics N_M, T_D, T_{MR}, T_M, λ_M, P_A, and T_G. But in this form it is hard to capture and compare the performance of the different algorithms. For this reason the following two opposed metrics are singled out and presented graphically.

These two metrics are [CTA00]:

λ_M: This measures the numbers of wrong suspicions per second.

T_D: This is the average time that elapses since the crash of q until p starts to suspect q permanently.

The counteraction of this two metrics is obvious. If the failure detector's tuning parameter is adjusted in a way to detect failures fast, the number of wrong suspicions is likely to be high. If the other way round the tuning parameter is set in a less aggressive way, the number of wrong suspicions will decrease at the cost of a longer detection time in the case of a failure.

The results of the performance measurements of Experiment 1.1 - 1.12 are depicted in Figure 2.17 - 2.28. These figures show the average detection time T_D on the horizontal axis and the mistake rate λ_M on the vertical axis. Values near to the lower left corner represent a short detection time with few mistakes. The basic failure detection algorithm of this work, as presented in Algorithm 1, is denoted with `basic`. Chen's failure detection algorithm [CTA00] is denoted with `Chen`, the φ failure detector of Hayashibara et al. [HDYK04] with φ, and Bertier's algorithm [BMS02] with `Bertier`.

Every single point of the graph of the basic failure detection algorithm and the algorithms of Chen et al. and Hayashibara et al. represents the result of one run of the test centre with different tuning parameters. As Bertier's failure detector has no tuning parameters it is only one single point in the charts.

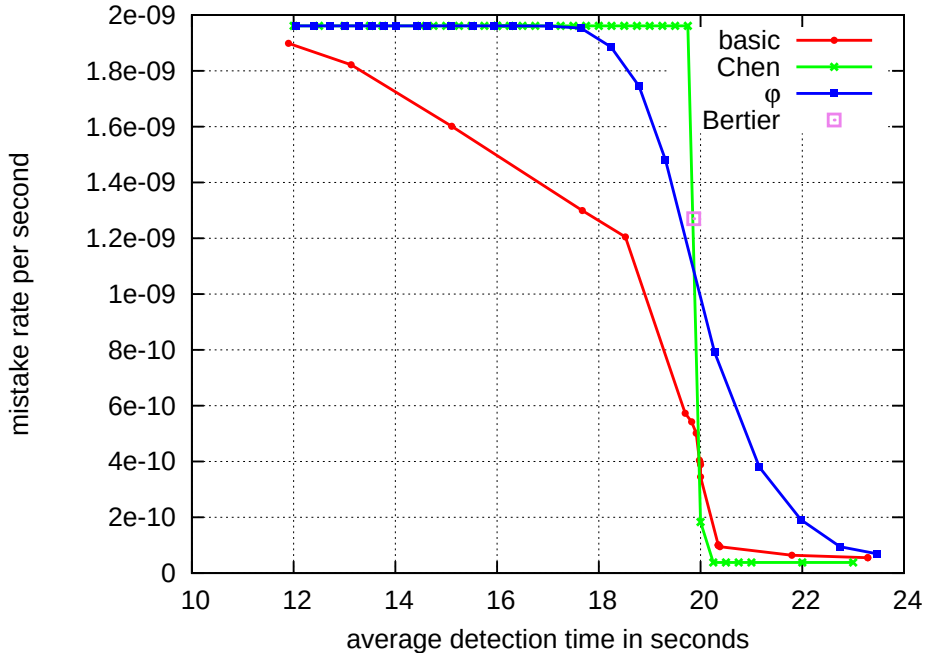


Figure 2.17: *Experiment 1.1: η: 1000, χ: 2%, κ: 1*

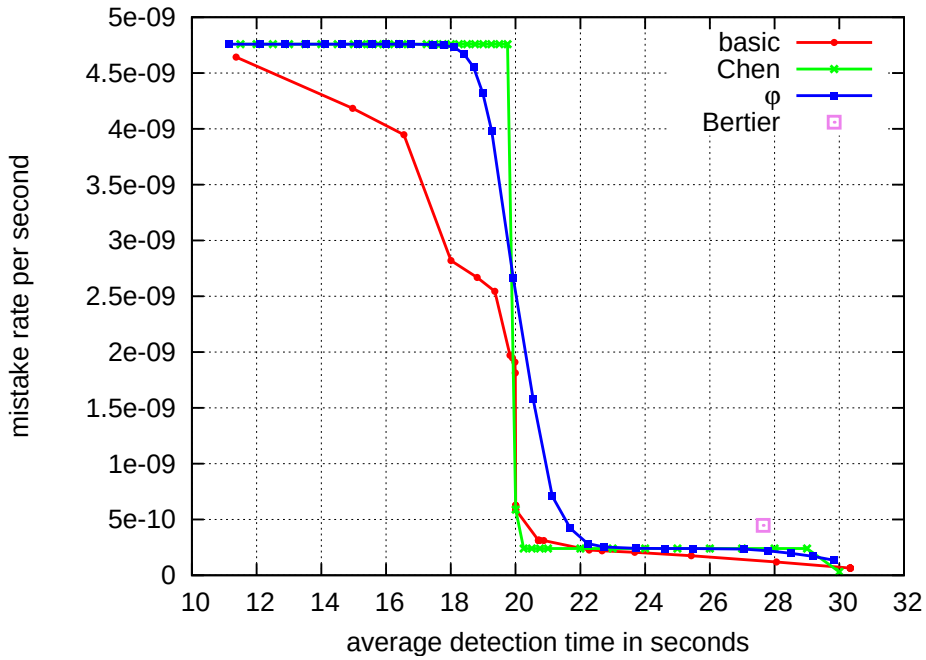


Figure 2.18: *Experiment 1.2: η: 1000, χ: 5%, κ: 1*

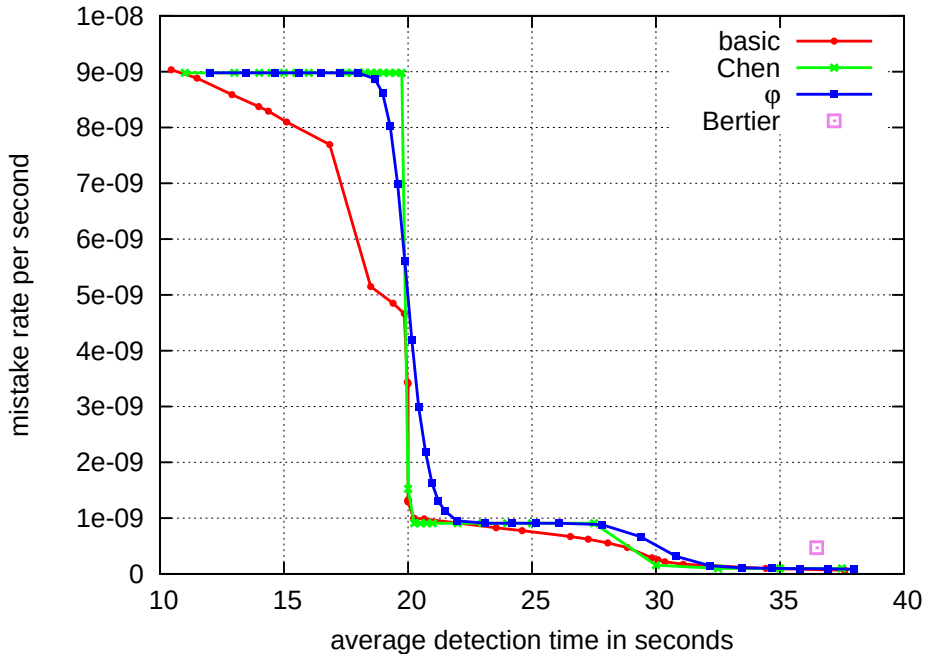


Figure 2.19: *Experiment 1.3: η: 1000, χ: 10%, κ: 1*

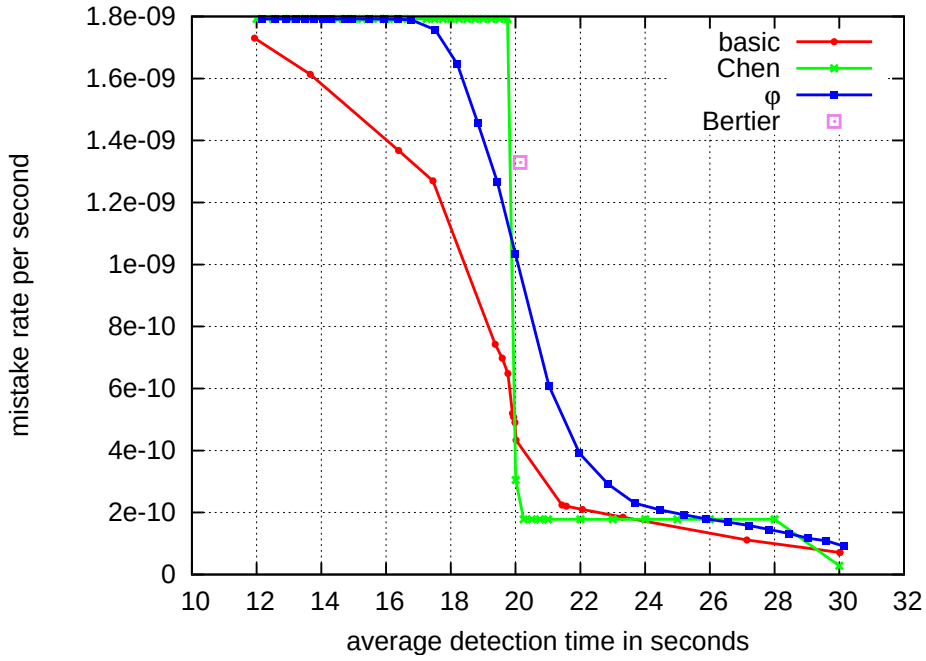


Figure 2.20: *Experiment 1.4: η: 1000, χ: 2%, κ: 5*

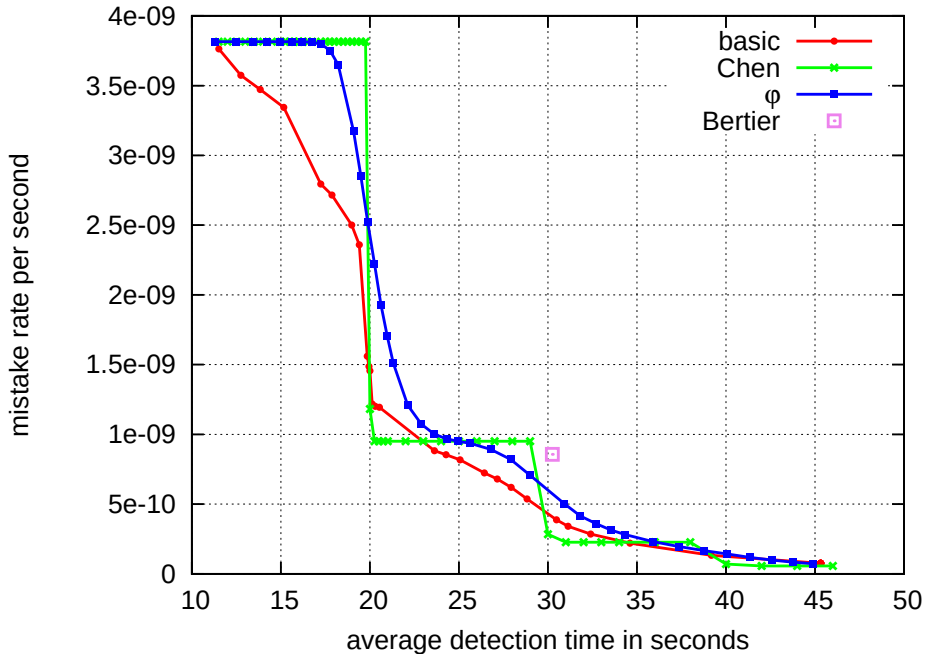


Figure 2.21: *Experiment 1.5: η: 1000, χ: 5%, κ: 5*

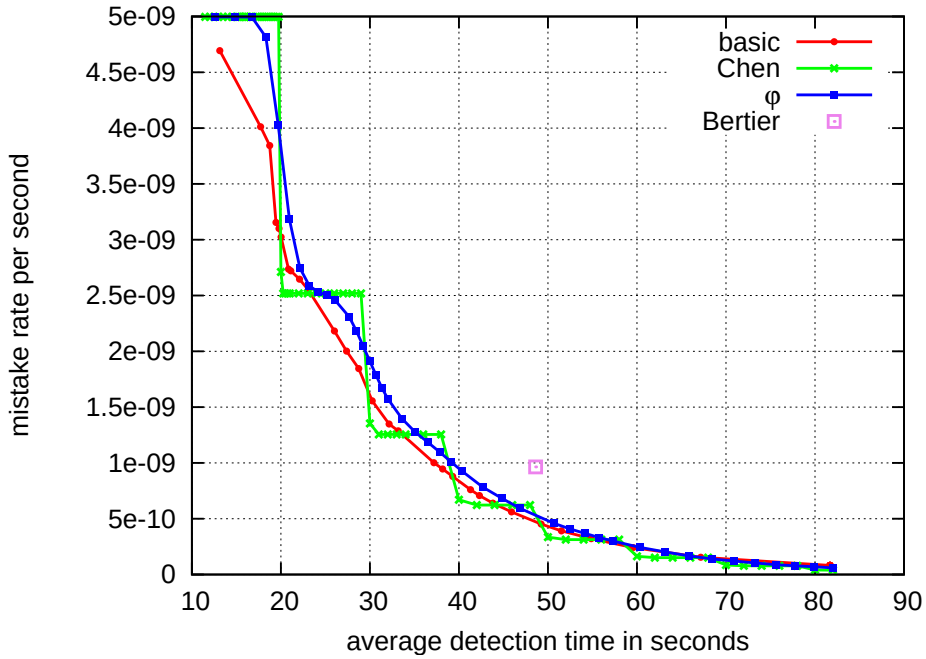


Figure 2.22: *Experiment 1.6: η: 1000, χ: 10%, κ: 5*

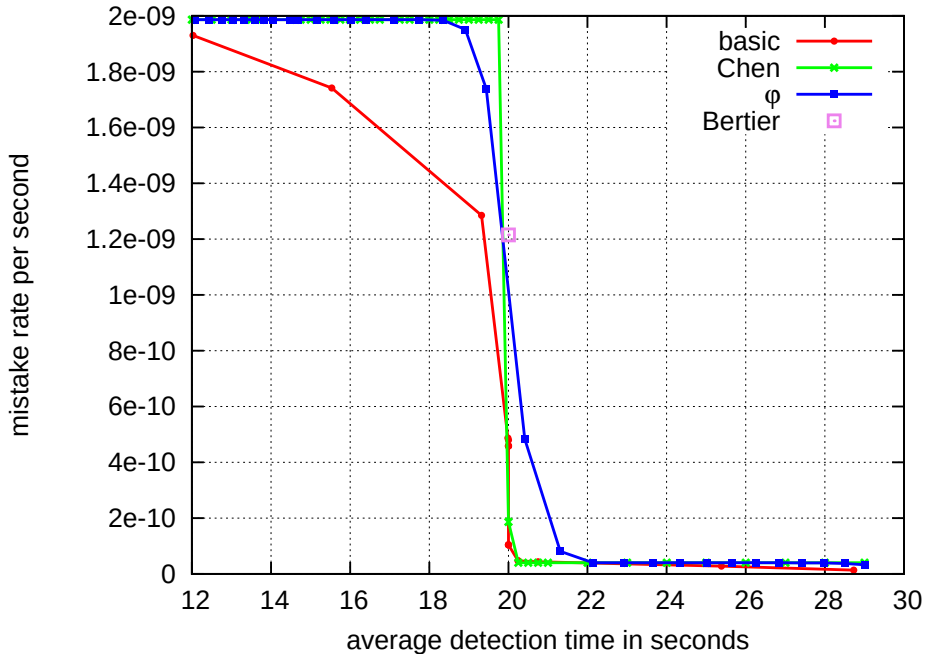


Figure 2.23: *Experiment 1.7: η: 20000, χ: 2%, κ: 1*

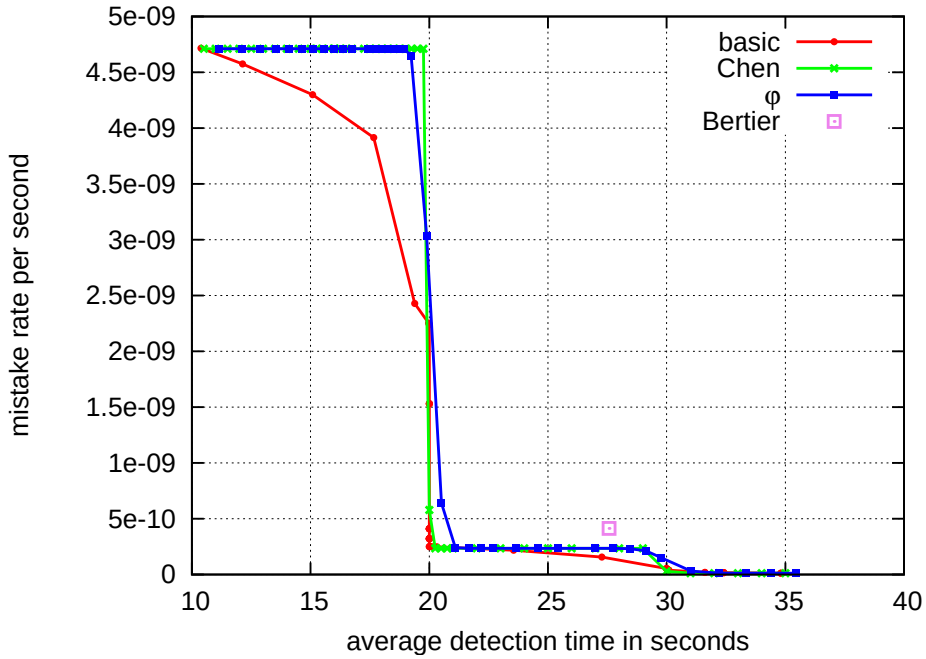


Figure 2.24: *Experiment 1.8: η: 20000, χ: 5%, κ: 1*

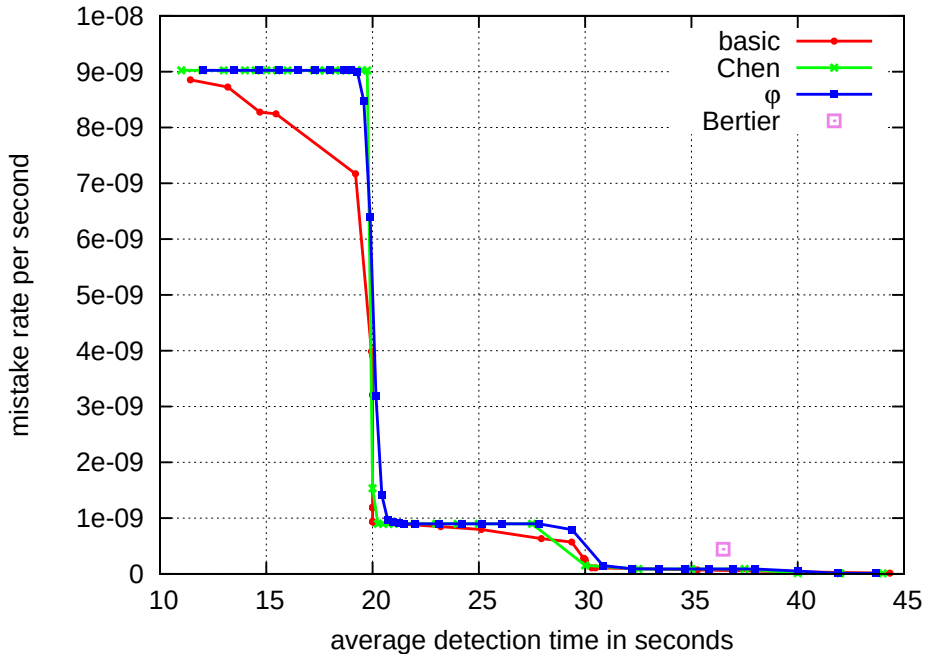


Figure 2.25: *Experiment 1.9: η: 20000, χ: 10%, κ: 1*

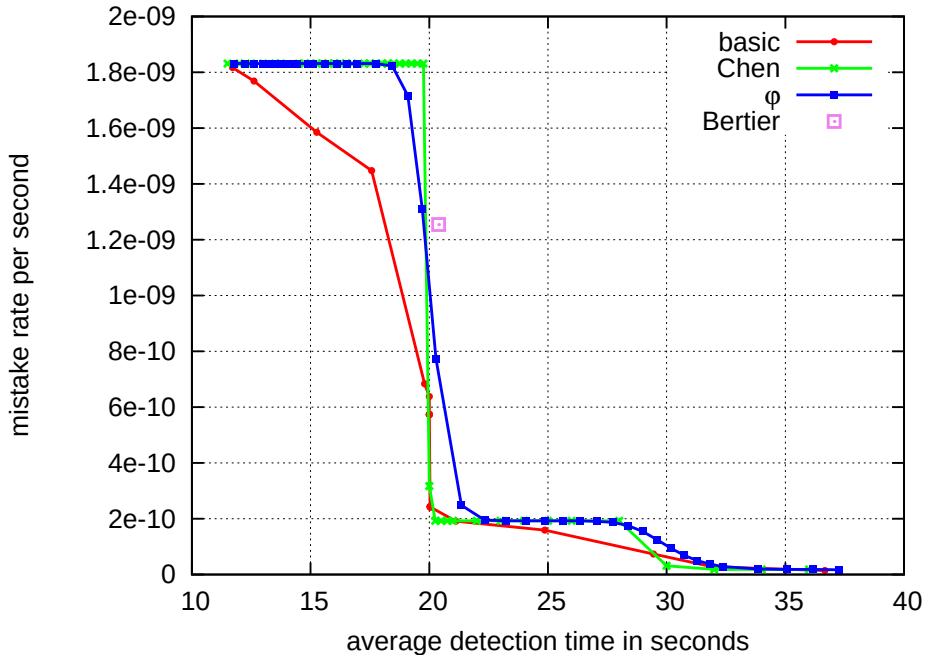


Figure 2.26: *Experiment 1.10: η: 20000, χ: 2%, κ: 5*

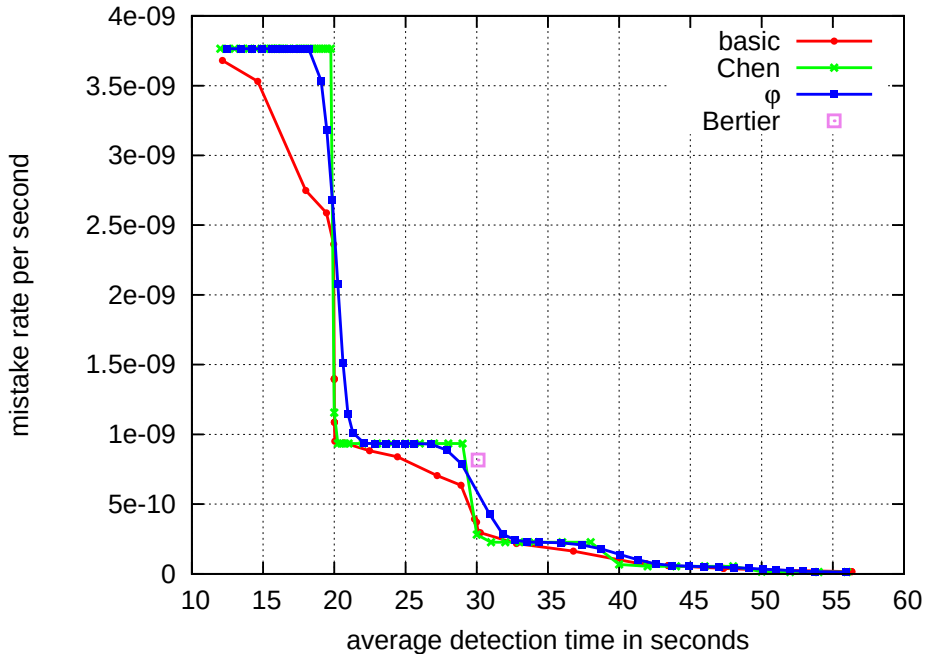


Figure 2.27: *Experiment 1.11: η: 20000, χ: 5%, κ: 5*

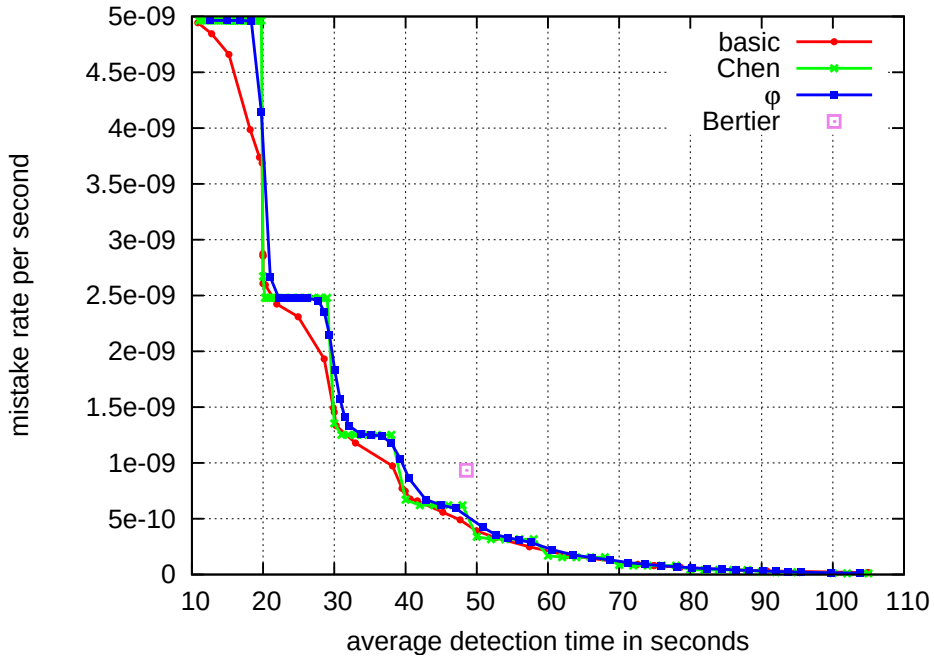


Figure 2.28: *Experiment 1.12: η: 20000, χ: 10%, κ: 5*

Basic algorithm vs. variations In the same manner as the comparison of the basic failure detection algorithm with Chen's, Bertier's, and Hayashibara's failure detectors, the basic failure detection algorithm is compared to the variations proposed in 2.5.

The following five algorithms have been investigated within these experiments:

basic: The basic failure detection algorithm as in the previous experiments.

send: A variation of the basic failure detection algorithm using the different freshness point strategy (see Section 2.5.3).

adjust: This variation implements the self-adjusting features as described in Section 2.5.2.

send+adjust: A combination of the different freshness point strategy and the self-adjusting variation.

smooth: A variation of the basic failure detection algorithm using the histogram smoothing technique with 100 bins and the smoothing parameter k set to 2 (see Section 2.5.4).

For these failure detectors, experiments with the same settings as above have been conducted:

Experiment 2.1: η: 1000, χ: 2%, κ: 1	(Figure 2.29)
Experiment 2.2: η: 1000, χ: 5%, κ: 1	(Figure 2.30)
Experiment 2.3: η: 1000, χ: 10%, κ: 1	(Figure 2.31)
Experiment 2.4: η: 1000, χ: 2%, κ: 5	(Figure 2.32)
Experiment 2.5: η: 1000, χ: 5%, κ: 5	(Figure 2.33)
Experiment 2.6: η: 1000, χ: 10%, κ: 5	(Figure 2.34)
Experiment 2.7: η: 20000, χ: 2%, κ: 1	(Figure 2.35)
Experiment 2.8: η: 20000, χ: 5%, κ: 1	(Figure 2.36)
Experiment 2.9: η: 20000, χ: 10%, κ: 1	(Figure 2.37)
Experiment 2.10: η: 20000, χ: 2%, κ: 5	(Figure 2.38)
Experiment 2.11: η: 20000, χ: 5%, κ: 5	(Figure 2.39)
Experiment 2.12: η: 20000, χ: 10%, κ: 5	(Figure 2.40)

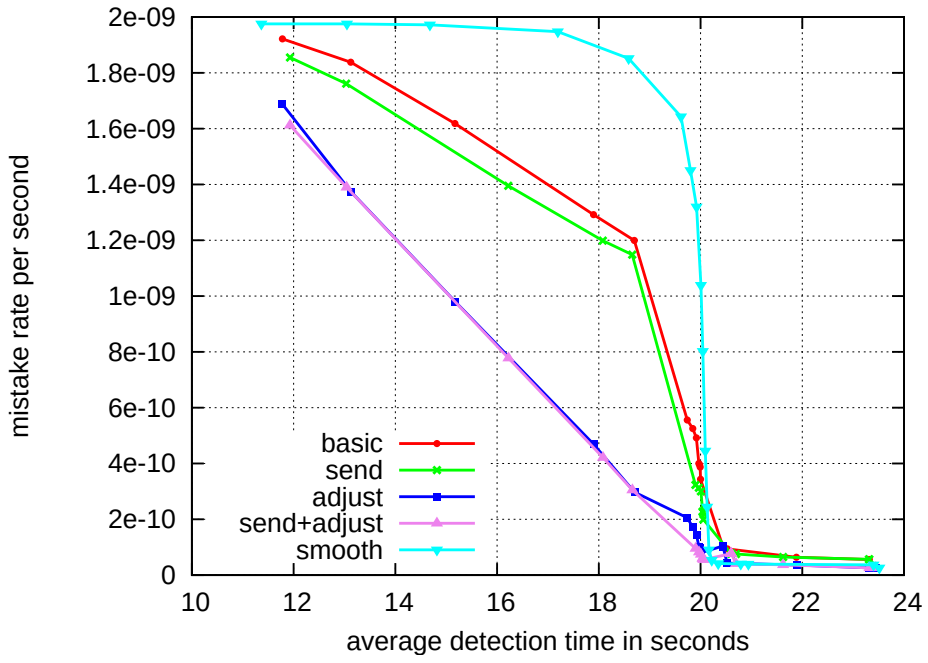


Figure 2.29: *Experiment 2.1: η: 1000, χ: 2%, κ: 1*

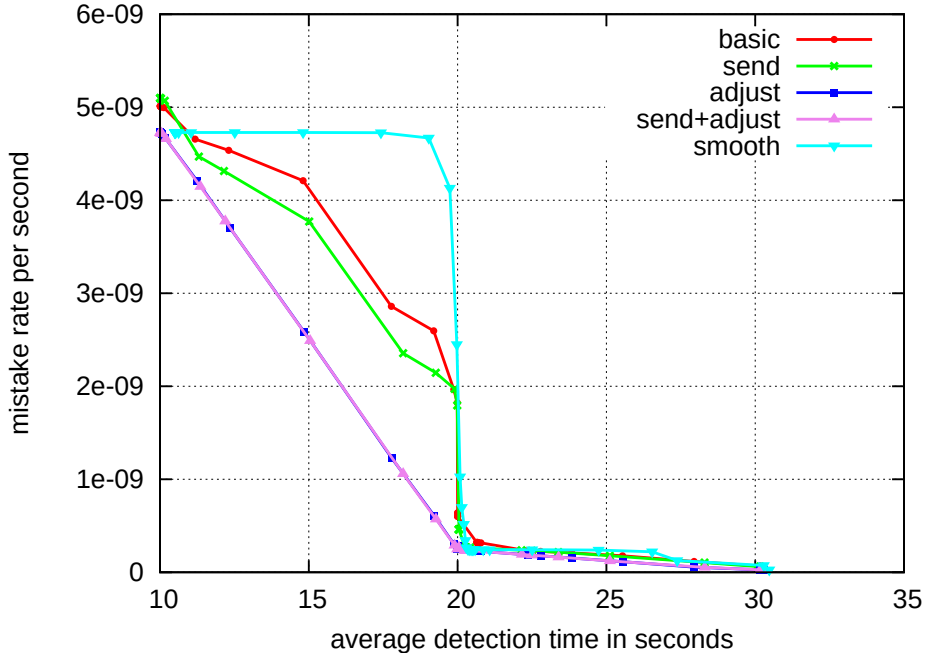


Figure 2.30: *Experiment 2.2: η: 1000, χ: 5%, κ: 1*

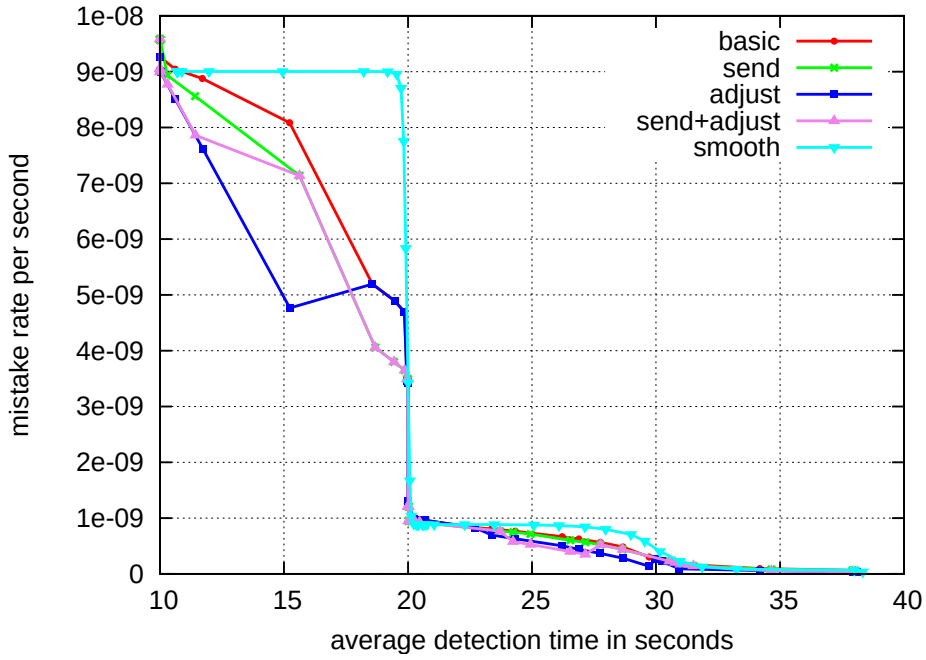


Figure 2.31: *Experiment 2.3: η: 1000, χ: 10%, κ: 1*

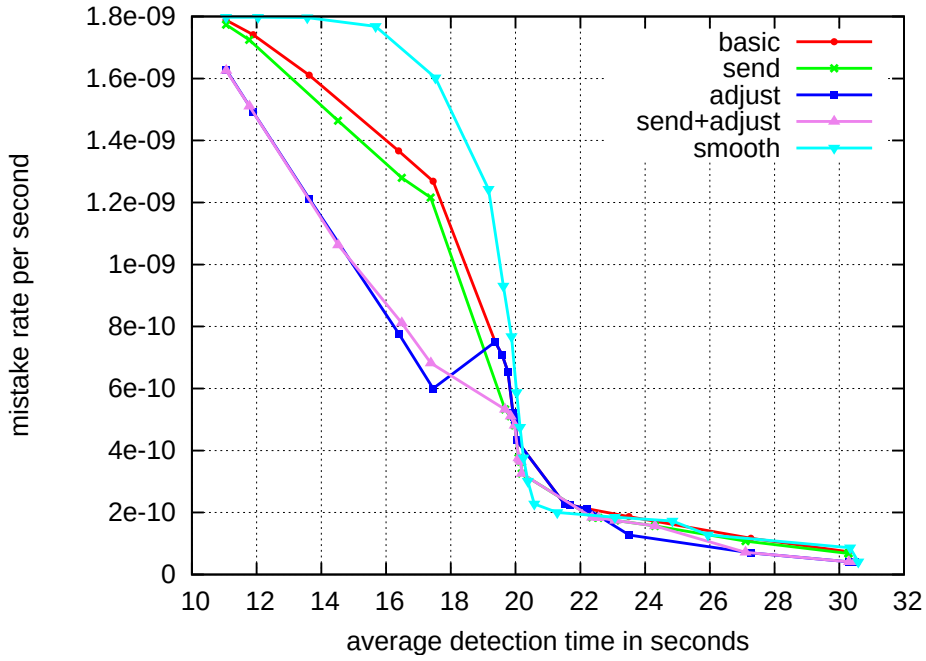


Figure 2.32: *Experiment 2.4: η: 1000, χ: 2%, κ: 5*

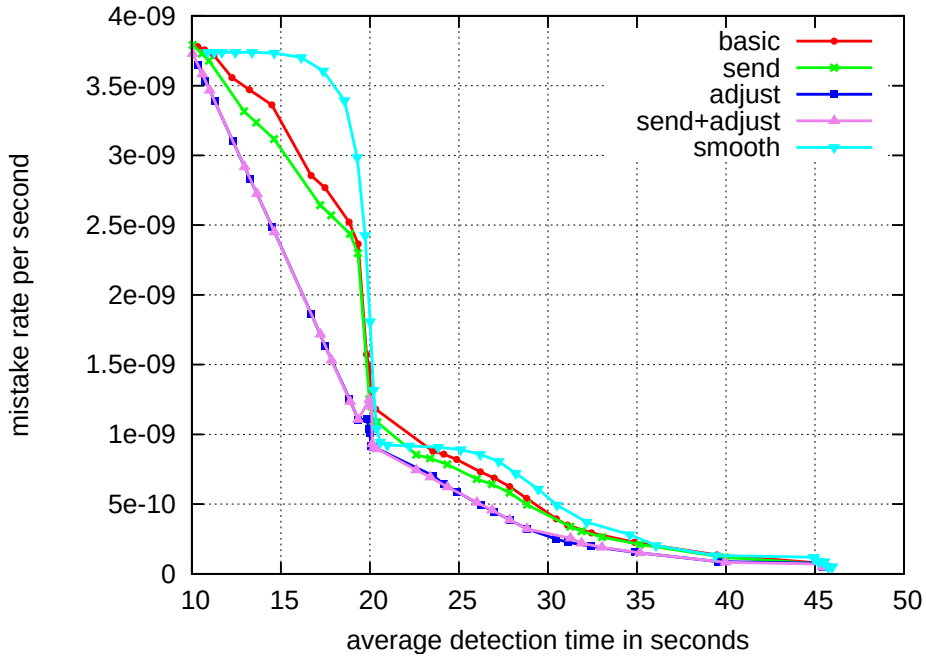


Figure 2.33: *Experiment 2.5: η: 1000, χ: 5%, κ: 5*

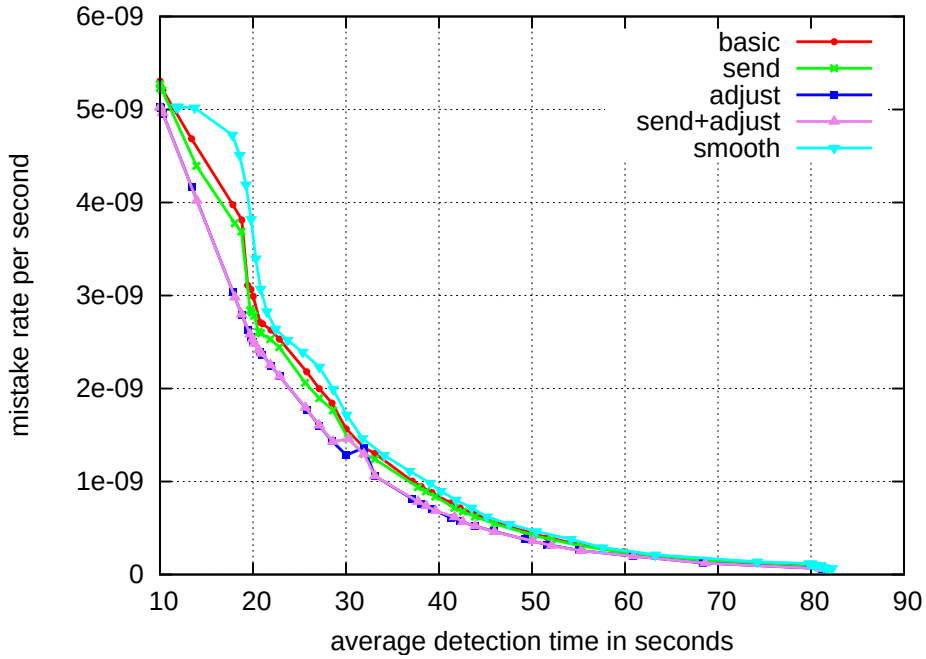


Figure 2.34: *Experiment 2.6: η: 1000, χ: 10%, κ: 5*

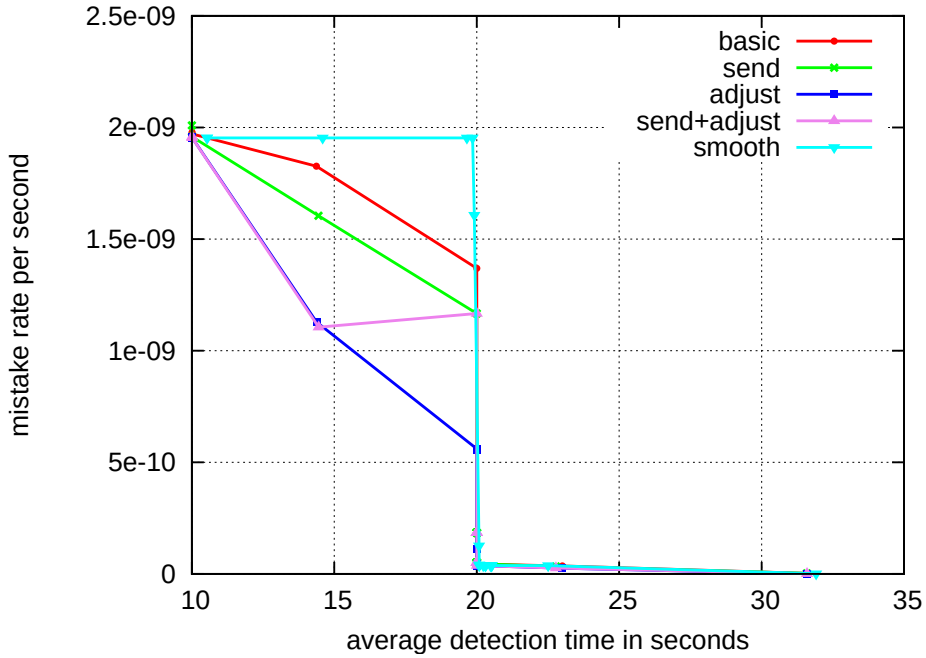


Figure 2.35: *Experiment 2.7: η: 20000, χ: 2%, κ: 1*

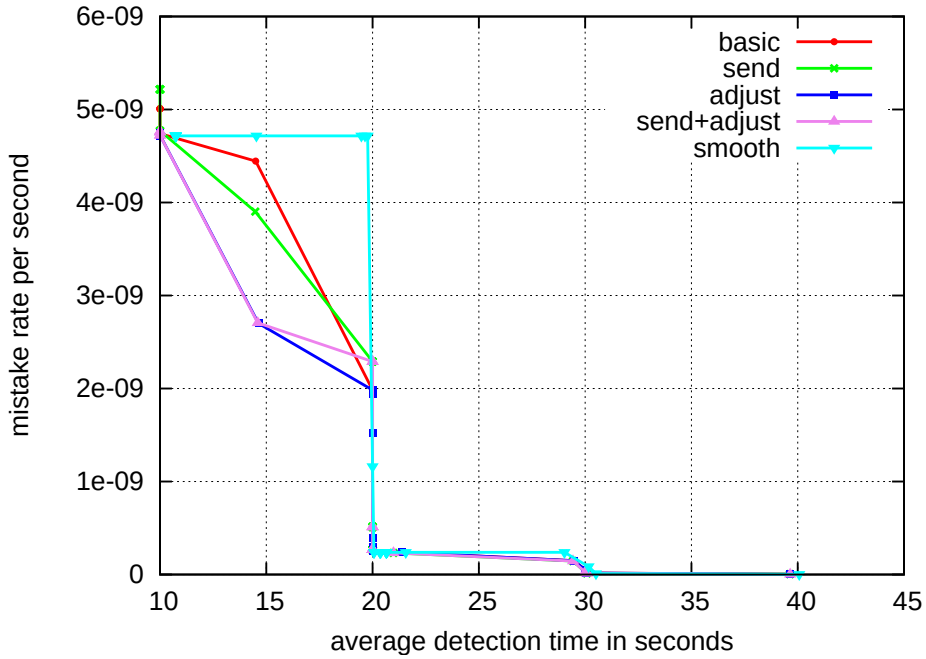


Figure 2.36: *Experiment 2.8: η: 20000, χ: 5%, κ: 1*

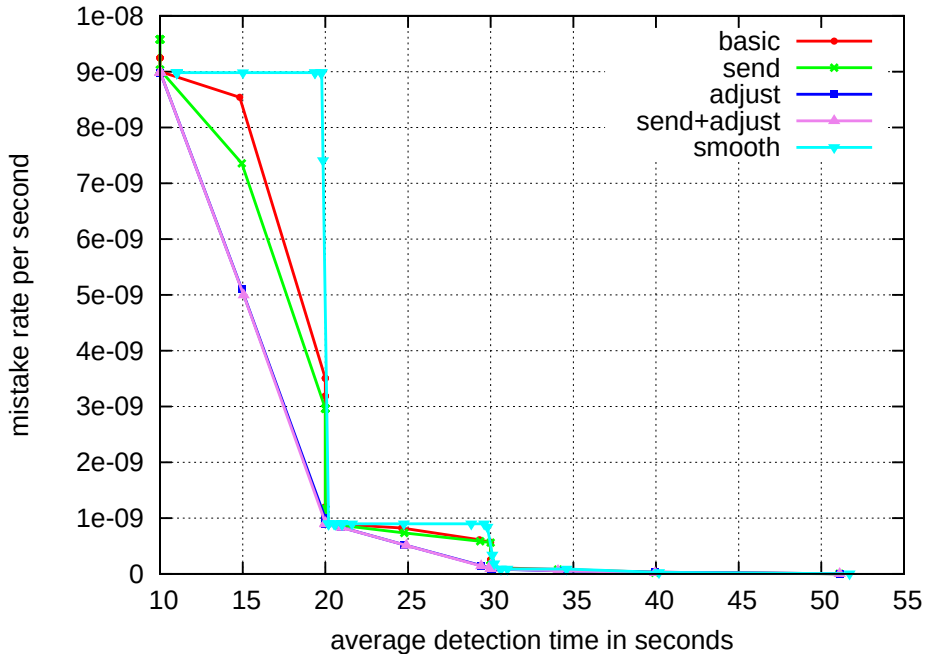


Figure 2.37: *Experiment 2.9: η: 20000, χ: 10%, κ: 1*

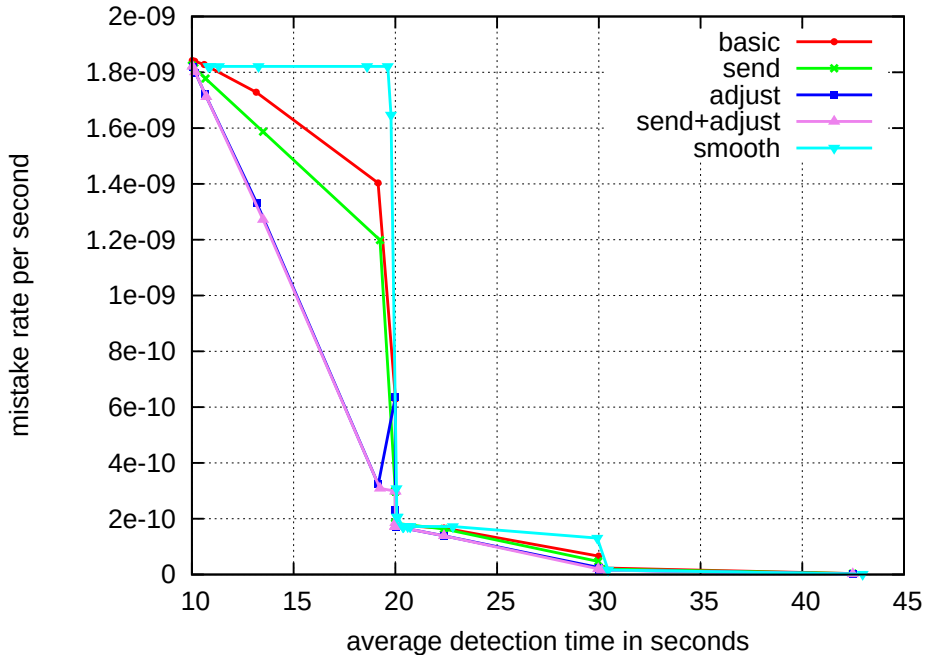


Figure 2.38: *Experiment 2.10: η: 20000, χ: 2%, κ: 5*

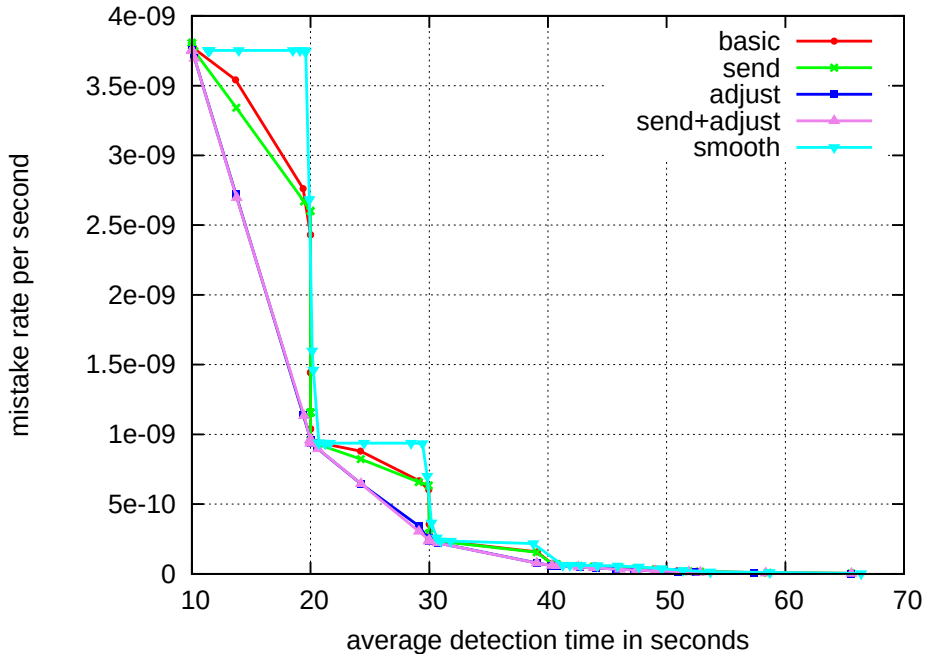


Figure 2.39: *Experiment 2.11: η: 20000, χ: 5%, κ: 5*

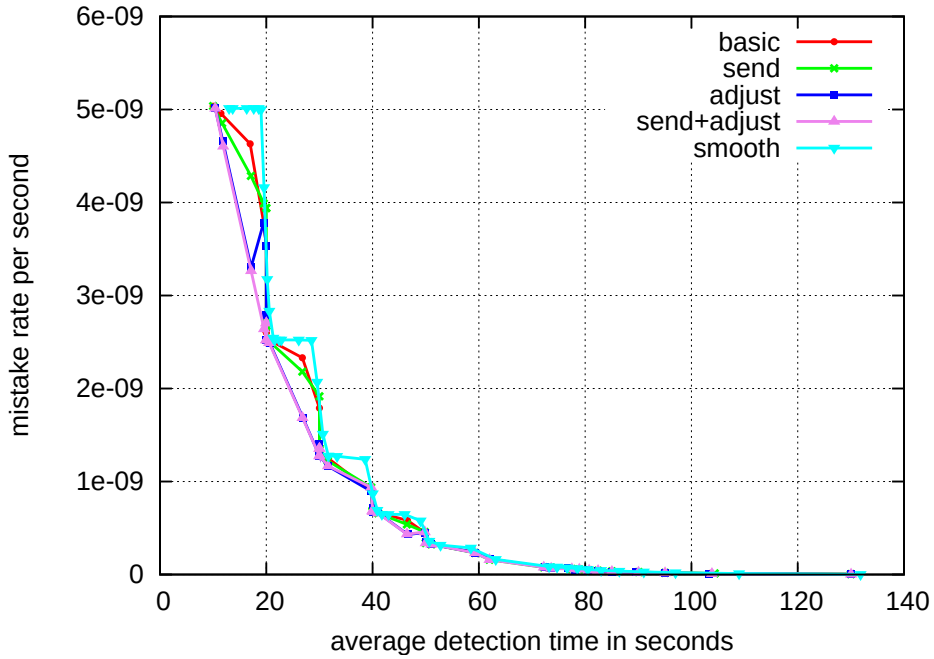


Figure 2.40: *Experiment 2.12: η: 20000, χ: 10%, κ: 5*

Send+adjust vs. other state of the art algorithms Experiment 1.1 - 1.12 compare the basic failure detection algorithm developed in this work to other state of the art failure detectors. Experiment 2.1 - 2.12 compare the basic failure detection algorithm to algorithms implementing the proposed extensions. In most settings the variant `send+adjust` performed best, also better than `basic`. Therefore it is interesting to compare `send+adjust` to the other state of the art algorithms: `Chen`, φ, and `Bertier`. This is done in the following Experiment 3.1 - 3.12, with an analogue setting as in Experiment 1.1 - 1.12 and Experiment 2.1 - 2.12.

Experiment 3.1: η: 1000, χ: 2%, κ: 1	(Figure 2.41)
Experiment 3.2: η: 1000, χ: 5%, κ: 1	(Figure 2.42)
Experiment 3.3: η: 1000, χ: 10%, κ: 1	(Figure 2.43)
Experiment 3.4: η: 1000, χ: 2%, κ: 5	(Figure 2.44)
Experiment 3.5: η: 1000, χ: 5%, κ: 5	(Figure 2.45)
Experiment 3.6: η: 1000, χ: 10%, κ: 5	(Figure 2.46)
Experiment 3.7: η: 20000, χ: 2%, κ: 1	(Figure 2.47)
Experiment 3.8: η: 20000, χ: 5%, κ: 1	(Figure 2.48)
Experiment 3.9: η: 20000, χ: 10%, κ: 1	(Figure 2.49)
Experiment 3.10: η: 20000, χ: 2%, κ: 5	(Figure 2.50)
Experiment 3.11: η: 20000, χ: 5%, κ: 5	(Figure 2.51)
Experiment 3.12: η: 20000, χ: 10%, κ: 5	(Figure 2.52)

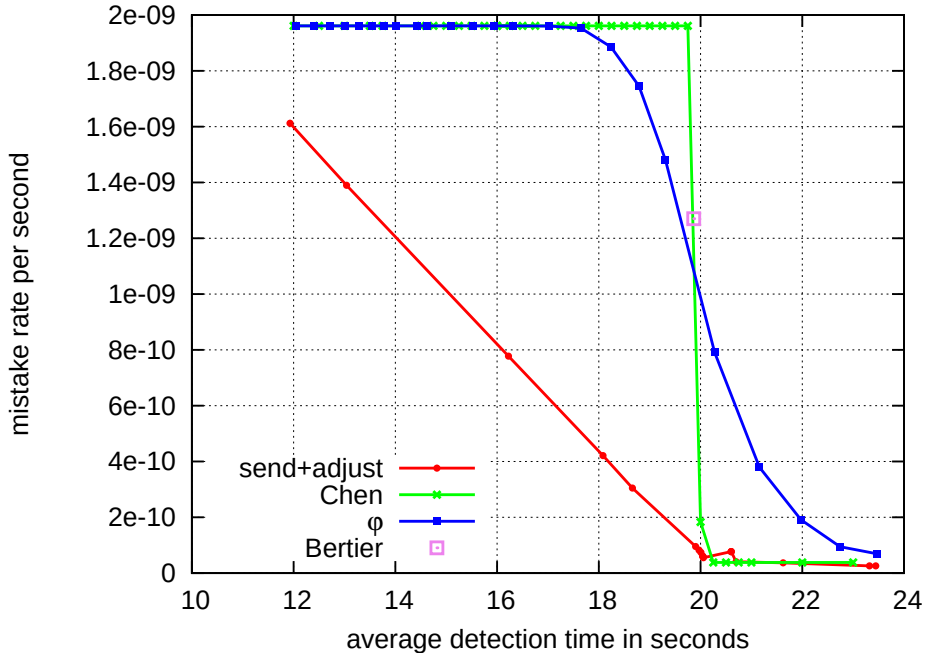


Figure 2.41: *Experiment 3.1: η: 1000, χ: 2%, κ: 1*

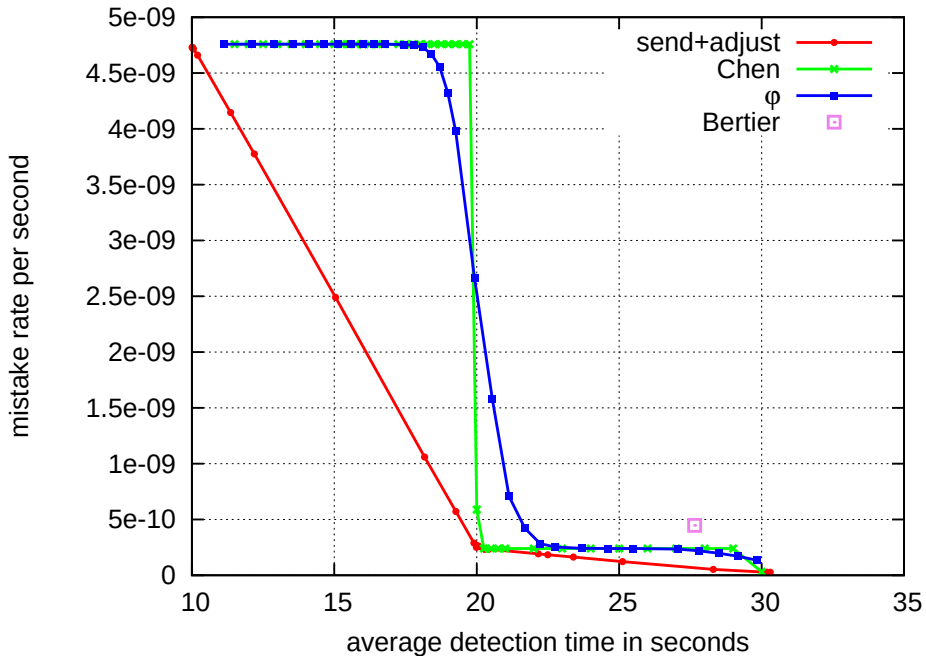


Figure 2.42: *Experiment 3.2: η: 1000, χ: 5%, κ: 1*

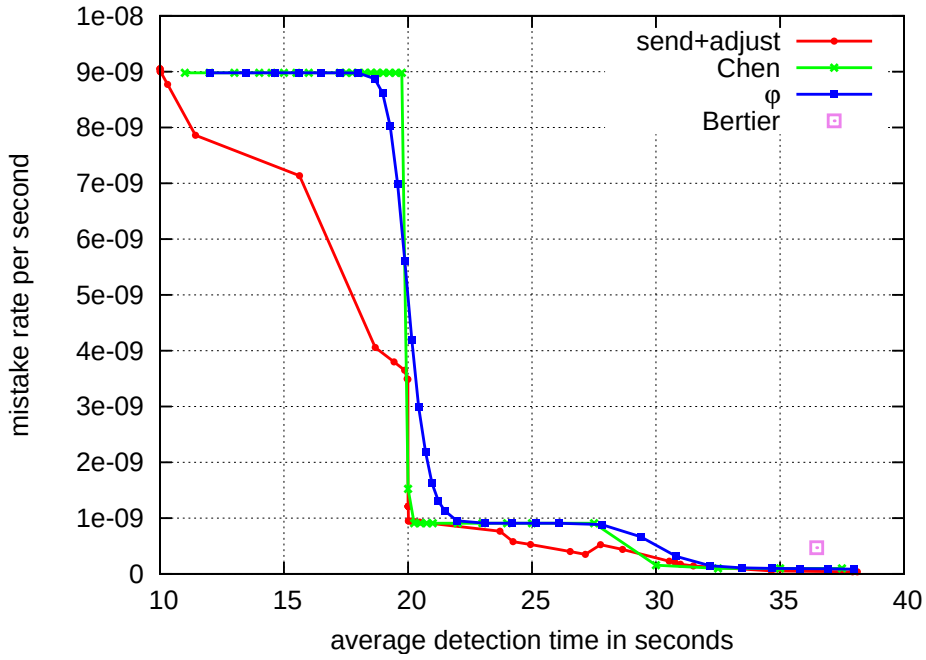


Figure 2.43: *Experiment 3.3: η: 1000, χ: 10%, κ: 1*

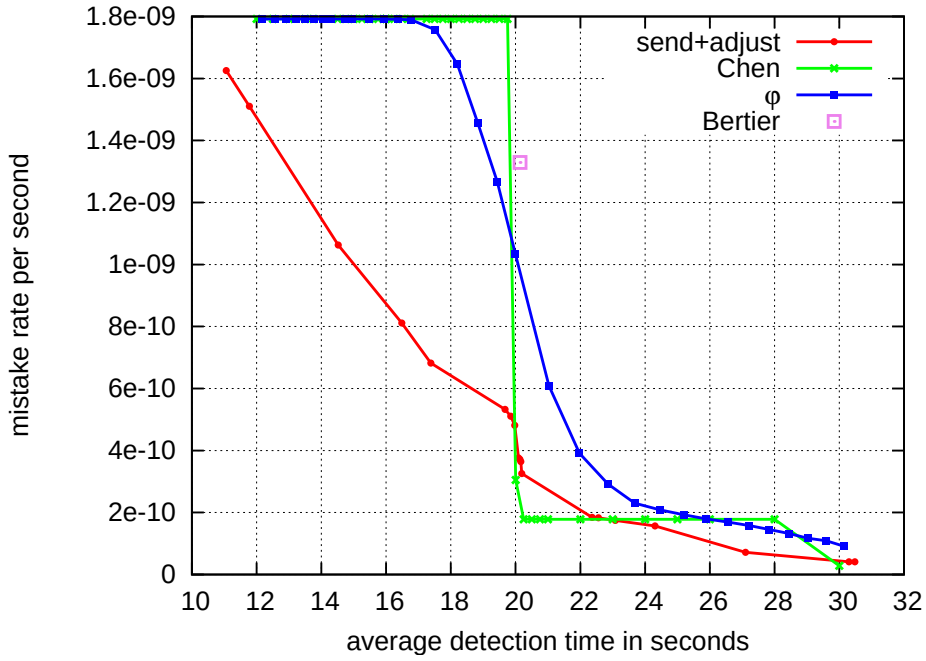


Figure 2.44: *Experiment 3.4: η: 1000, χ: 2%, κ: 5*

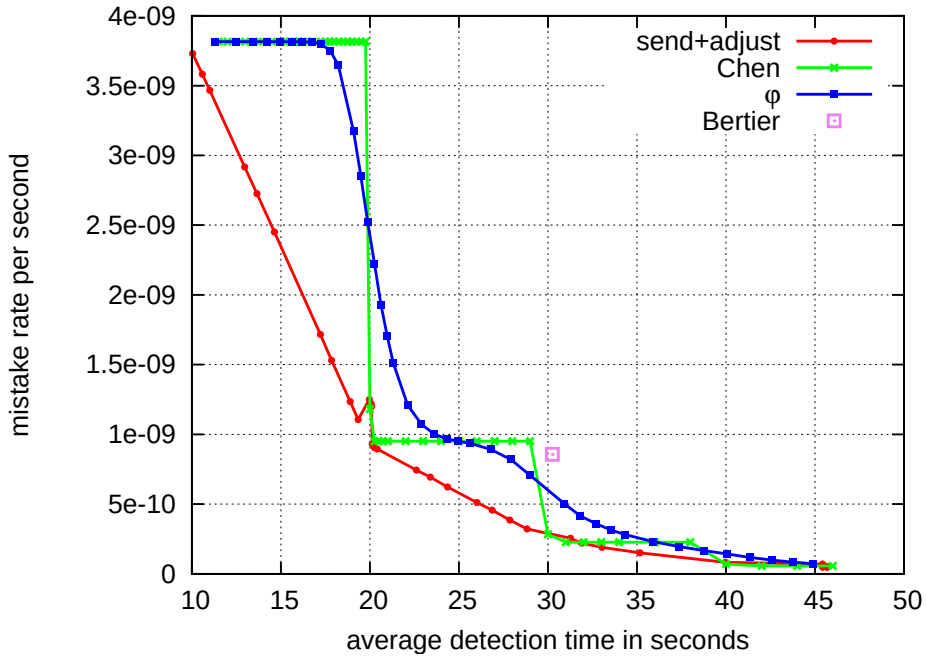


Figure 2.45: *Experiment 3.5: η: 1000, χ: 5%, κ: 5*

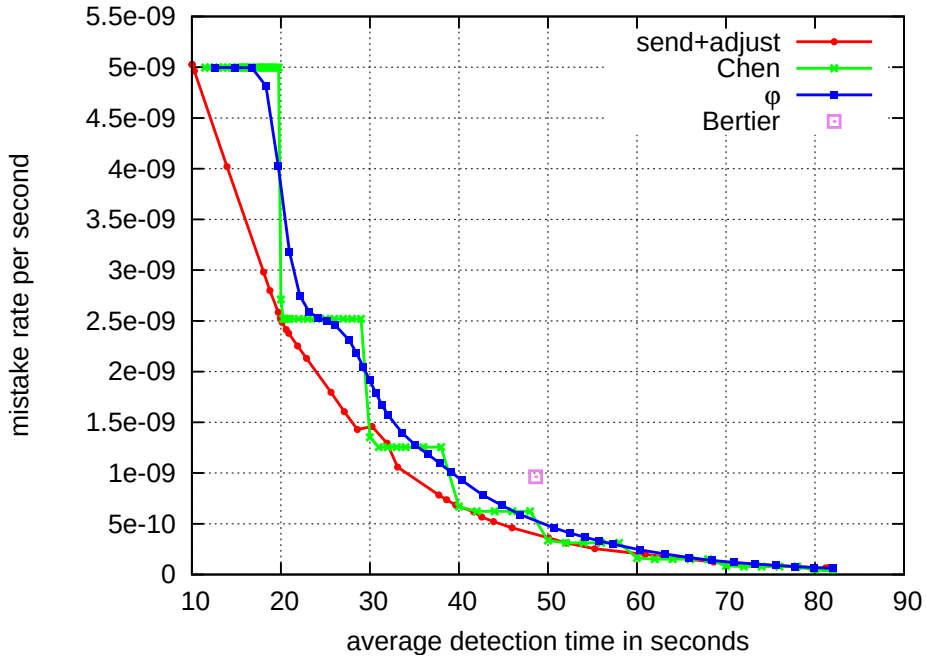


Figure 2.46: *Experiment 3.6: η: 1000, χ: 10%, κ: 5*

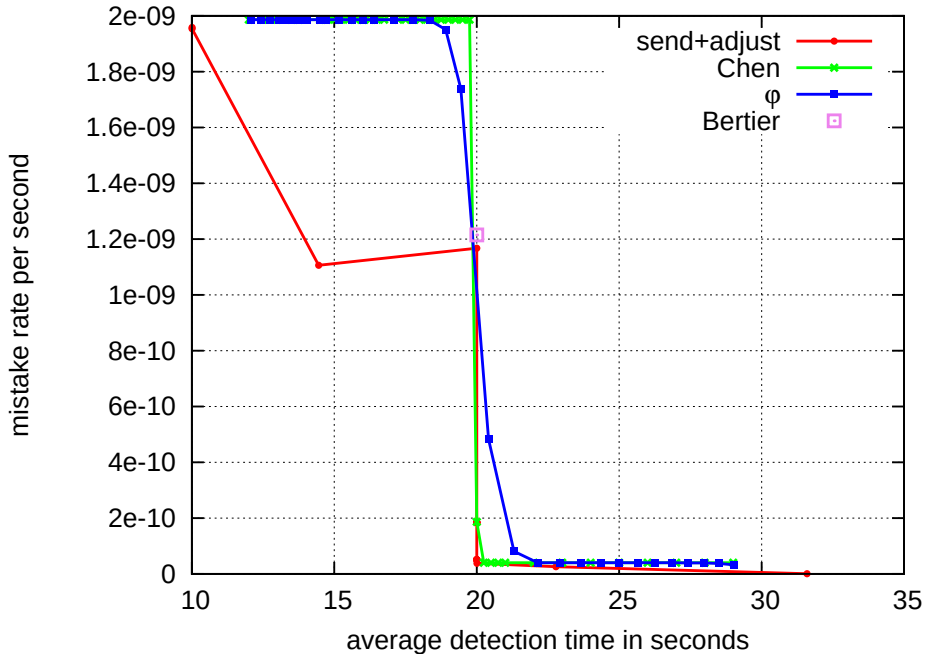


Figure 2.47: *Experiment 3.7: η: 20000, χ: 2%, κ: 1*

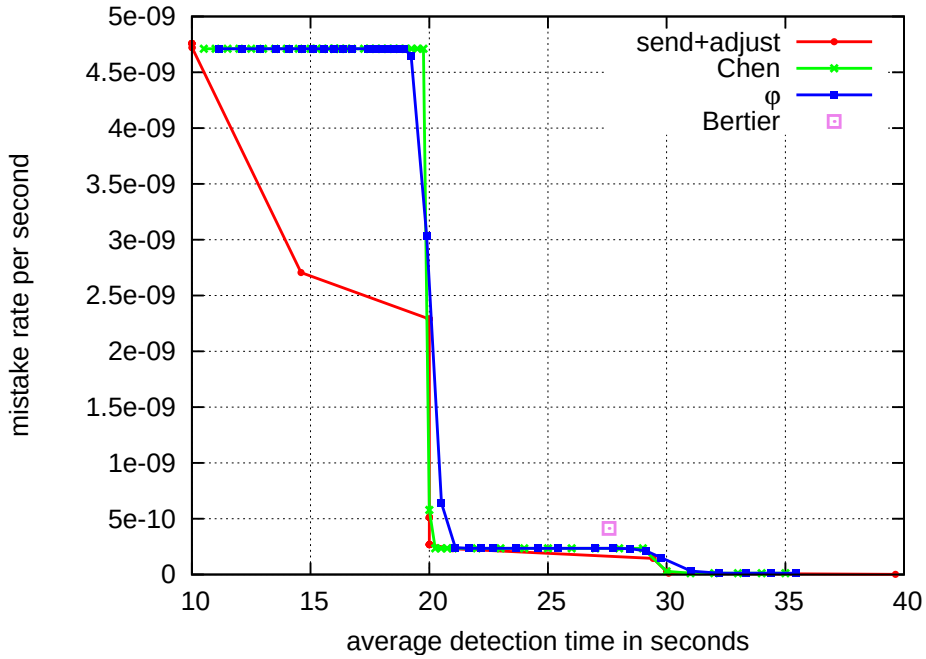


Figure 2.48: *Experiment 3.8: η: 20000, χ: 5%, κ: 1*

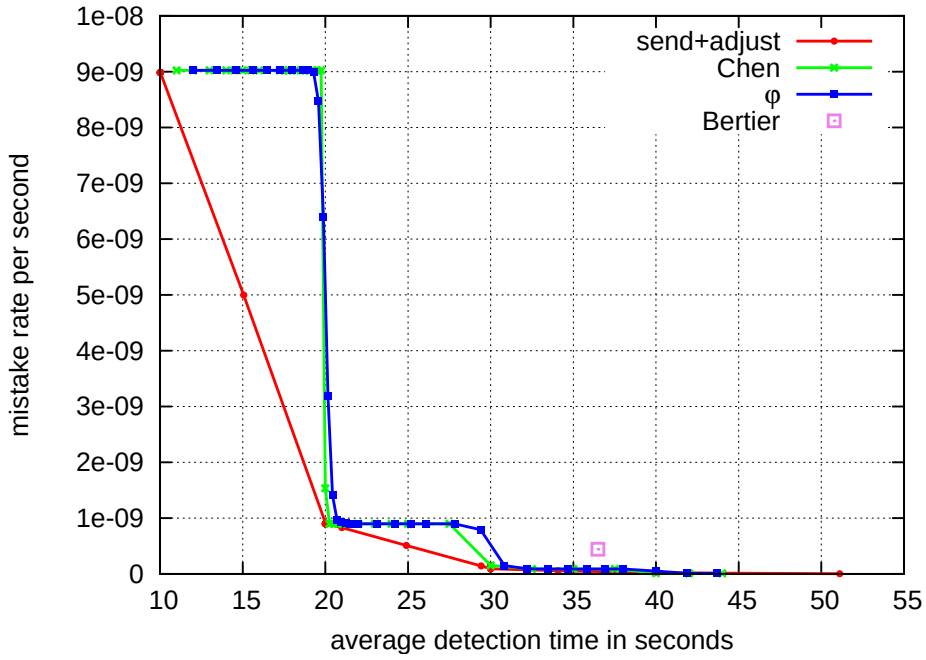


Figure 2.49: *Experiment 3.9: η: 20000, χ: 10%, κ: 1*

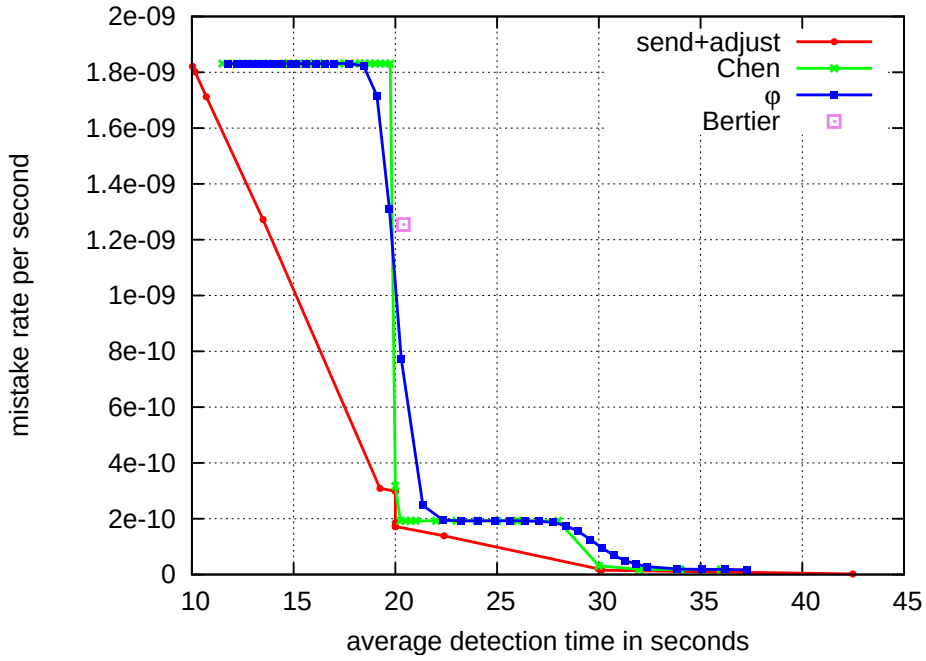


Figure 2.50: *Experiment 3.10: η: 20000, χ: 2%, κ: 5*

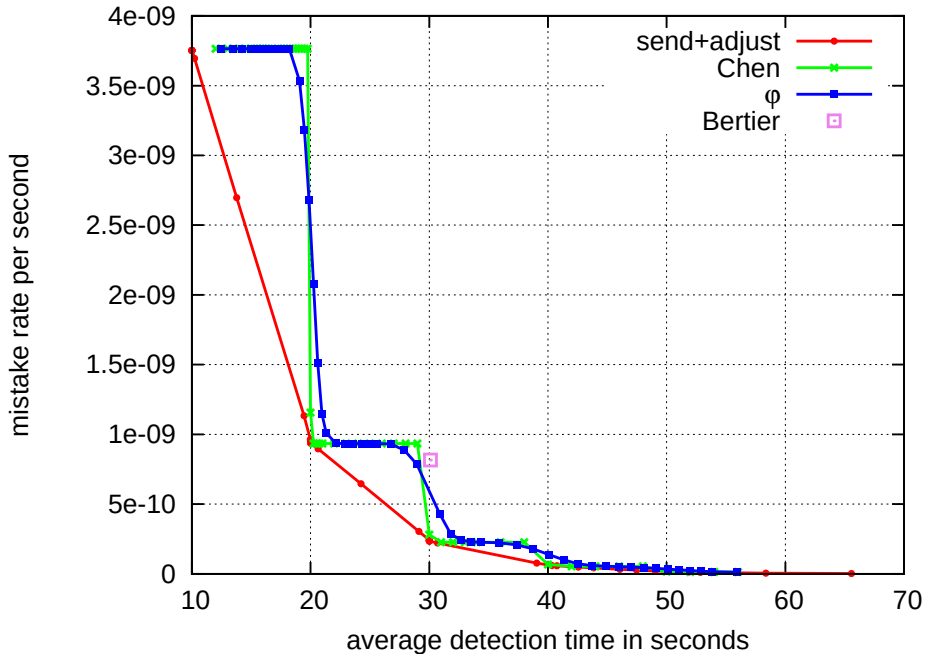


Figure 2.51: *Experiment 3.11: η: 20000, χ: 5%, κ: 5*

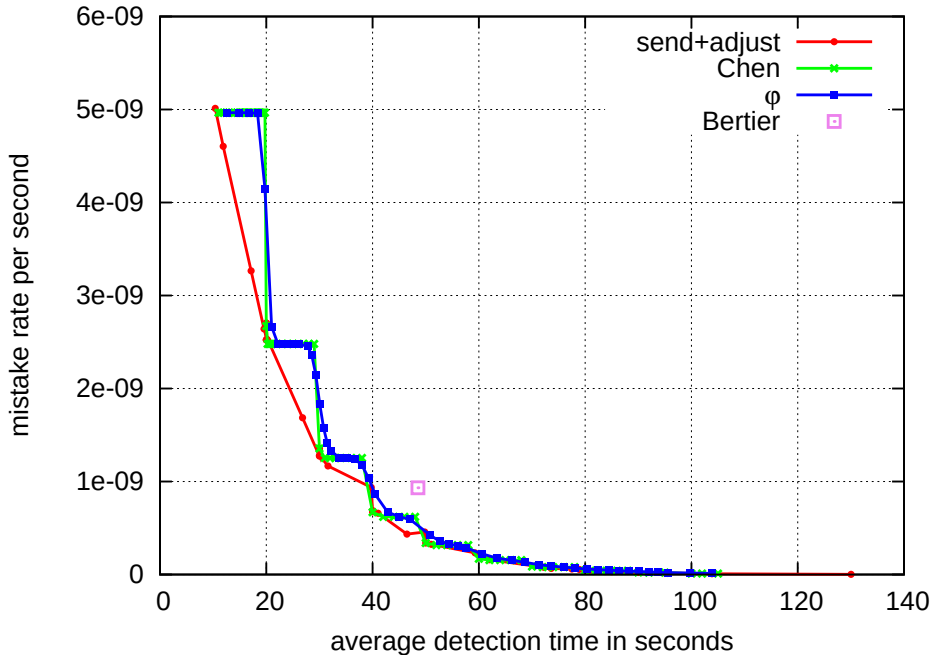


Figure 2.52: *Experiment 3.12: η: 20000, χ: 10%, κ: 5*

2.6.3 Discussion of the evaluation results

Experiment 1.1 - 1.12 attest the basic failure detection algorithm a continuous good performance compared to the other algorithms. The basic failure detection algorithm achieves excellent results with all kinds of message losses. Moreover, for longer sampling windows the small areas where other algorithms could outperform the basic algorithm further shrink.

Besides its performance the basic algorithm has further advantages. It is more flexible than Chen's, Bertier's, and other non-accrual failure detectors. The accrual φ failure detector is restricted to environments with roughly normal distributed heartbeat inter-arrival times. The failure detectors proposed in this work do not have this limitation as the probability distribution of the heartbeat-arrivals is estimated in a non-model-based manner. This further emphasises the flexibility of the introduced algorithm.

Experiment 2.1 - 2.12 show that all investigated variations except the one based on histogram smoothing indeed provide an even better performance than the basic algorithm. The variation that uses a different freshness point strategy to abolish the dependence on the last heartbeats performs nearly in all cases better than the basic algorithm.

The self-adjusting variants show at least the same and mostly better results than its non-self-adjusting counterparts. The algorithm `adjust` performed always better or equal than the `basic` algorithm, the algorithm `send+adjust` better or equal than `send`.

Experiment 3.1 - 3.12 show that the variation `send+adjust` of the failure detector proposed in this work clearly outperforms the other state of the art failure detection algorithms in all settings, more clearly than `basic` does in Experiment 1.1 - 1.12.

To sum up, the failure detector developed in this work outperforms all other investigated failure detectors in all investigated test scenarios. In Experiment 3.1 for instance (see Figure 2.41) it can be seen that in some cases the algorithm proposed in this work makes about 90% less wrong suspicions than the other algorithms with the same detection time.

In [SPTU07a], different scenarios than in this work have been chosen to compare the basic failure detection algorithm to Chen's, Hayashibara's, and Bertier's. In the case of the consideration of message loss the basic failure detection algorithm performed best, too. It is likely that the developed algorithms, especially the variation `send+adjust`, currently represent the best heartbeat-style failure detectors in the presence of message loss.

2.7 Lazy monitoring

Many failure detectors, like the ones proposed here, use heartbeats to draw conclusions about the state of processes/nodes within a distributed environment. The contribution of this section is an approach whose benefits are twofold. On the one hand it reduces the network overhead produced by heartbeat-style failure detectors. On the other hand it improves the quality of these failure detectors by providing them with richer information about the current network condition. This approach is called *lazy monitoring*, since the active sending of heartbeats is avoided if possible as introduced in Section 2.3.5. Because it is independent of the actual failure detection algorithm, it can be used in many domains. As this technique mainly aims at reducing the overhead of failure detectors it complements the introduced variations which focus on improving failure detection performance.

A heartbeat-style failure detector is defined to be *lazy* if it applies a technique to reduce the networking overhead that arises from sending heartbeat messages. The analogy is that these algorithms only send heartbeat messages if they really have to and are thus called lazy. In this context it is important to distinct application messages from heartbeat messages. The former are sent by the application and cannot be avoided, heartbeat messages are sent by failure detectors.

Fetzer et al. [FRT01] introduced the term lazy failure detector but came up with a slightly different definition which does not fit to heartbeat-style failure detectors very well. Their algorithm requires that each application message is being acknowledged. Thus the round trip delay of each application message together with its acknowledgement message is calculated. In addition, for each destination the maximum round trip delay is stored. The output of the failure detector depends on the existence of a pending message, i.e. a message such that the application message has been sent but not acknowledged yet: If there is no such message, the answer is “no suspect”, but a ping is sent to verify this answer. If there is such a message, the answer depends on the maximum round trip delay.

The lazy monitoring approach proposed in this work also uses application messages in order to save overhead, but has some noteworthy features in comparison to the failure detector of [FRT01]. The usage of application messages in order to save overhead is an integrated part of the failure detector of Fetzer et al. and is not applicable to other failure detection algorithms. The lazy monitoring approach proposed in this work can be seen as a form of a plug-in that can be used with heartbeat-style failure detectors. Fetzer’s algorithm uses application messages and is able to save in the optimal case 50% of its message overhead. It is based on computing round trip delays. Instead of sending ping-pong messages, with the use of application messages

it is able to save the "ping"-fraction. The lazy monitoring concept of this work is capable of saving nearly all the overhead produced by heartbeat-style failure detectors.

Larrea et al. [LLS+07] contribute to improve the communication efficiency of failure detectors. They assume a network of n processes forming a ring while the processes send heartbeats to their successor and monitor their predecessor by listening for heartbeats. Their focus is on reducing the number of unidirectional links between processes that carry messages forever. The authors of [LLS+07] propose to piggyback information, more precisely to append a suspicion list to sent heartbeats. By contrast, the lazy monitoring approach proposed here saves the heartbeats themselves and no possibly large data like a list is piggybacked.

The lazy monitoring technique presented in the following aims at using application messages as alive proof which is not novel. But so far, this fundamental concept cannot be used together with heartbeat-style failure detectors. These failure detectors completely depend on heartbeats sent at regular intervals and are incapable exploiting information application messages are providing about the environment. The main contribution of the lazy monitoring is to close this gap.

2.7.1 Lazy monitoring approach

Failure detectors should send as few messages as possible. There are two ways to influence the amount of sent heartbeat messages. First, heartbeat-style failure detectors are *tailorable*. This denotes the ability to downgrade the detection quality for the benefit of a lower network load. The heartbeat interval Δ_i is the key to tailor the failure detector in this way: A small heartbeat interval results in a high network load, but failures can be detected rapidly.

Algorithm 5 shows the traditional sampling method for heartbeat inter-arrival times. The monitored process q sends heartbeat messages to the monitoring process p every Δ_i. Process p manages a list S, the sample window with the sampled inter-arrival times, along with the freshness point f which is always set to the time when the last heartbeat was received and the heartbeat interval Δ_i. This information is needed by most failure detectors to compute a suspicion value. Whenever p receives a heartbeat, it appends $s = t_r - f$ (t_r is the current time) to S and sets $f = t_r$ afterwards. Most failure detectors limit the size of S to a certain value η (e.g. $\eta = 1000$) causing the oldest sample being deleted if a new one is inserted into S.

The lazy monitoring approach of this work aims at reducing the network load without the negative effects on the detection time. Quite the contrary,

Algorithm 5 Traditional heartbeat sampling

```
1:  Process q:
2:      send heartbeat message to p every Δ_i
3:
4:  Process p:
5:
6:      f = −1                                   ▷ freshness point
7:      S = nil                                  ▷ S is initialised as an empty list
8:
9:      procedure RCV_HB(m_j, t_r)               ▷ receiving heartbeat m_j at time t_r
10:         if f = −1 then
11:             f = t_r
12:         else
13:             s = t_r − f
14:             f = t_r
15:             append s to S
16:         end if
17:     end procedure
18:
```

it allows for a better training of the failure detector as it provides more data and thus can furthermore improve the quality of the generated suspicion information. Thereby it is distinguished between application messages and heartbeat messages. Application messages are messages sent by the members of the network in the normal mode and cannot be avoided. Heartbeat messages are the messages sent by failure detectors. This section aims at avoiding the overhead of sending heartbeats by appending small amounts of data to the application messages. First, it is assumed that application messages behave the same way as heartbeat messages. Limitations of this generalisation are discussed below.

Friedman et al. [FvR97] compare the throughput and latency of four protocols that provide total ordering. They come to the conclusion that message packing influences the performance overwhelmingly more than any other optimisation they check both in terms of throughput and latency. The reason is that packing messages reduces amongst others the header overhead for messages and the contention on the network. The lazy monitoring approach can be interpreted as some kind of message packing where the problem lies in enabling failure detectors to use them as samples. This is not possible yet for heartbeat-style failure detectors.

In order to save a heartbeat message, the lazy monitoring approach needs a consecutive id and a timestamp to be appended to an application message. In heartbeat-style failure detection algorithms, q sends a heartbeat to p every Δ_i seconds and p is sampling the inter-arrival times of these heartbeats.

Note that the calculation of inter-arrival times is always possible, provided p has a local clock. If the lazy monitoring approach is applied, process q's heartbeat sending behaviour must be slightly changed. A heartbeat is not sent automatically every Δ_i, but q is responsible to send one to p if no message, either application or heartbeat message, has been sent to p since Δ_i. Thus pure heartbeat messages are only used if no application message has been sent since Δ_i.

But a significant problem arises because the failure detector is now unable to compute new heartbeat inter-arrival times to insert into the sample window S. The sample window consists of heartbeat inter-arrival times which are typically around Δ_i. The inter-arrival times of application messages are arbitrary and have no informative value for a failure detector. Thus, the basic issue is to provide a technique that allows failure detectors to sample inter-arrival times although only application messages are sent at random times instead of heartbeats. Basically, heartbeat inter-arrival times are mainly influenced by the following three environmental circumstances:

Message delay: Heartbeats sent over the network are affected by message delays.

Message loss: It can occur that heartbeat messages get lost during the sending process.

Processing delay of q: The monitored process q sends heartbeat messages not at the time it is supposed, e.g. due to processing overload.

In many systems the variations of the inter-arrival times due to processing delays of q are negligible. Therefore, the focus here lies on message delay and message loss. In the following the concept of *lazy heartbeat sampling* is introduced. This enables p to sample heartbeat inter-arrival times in order to adapt to the actual network conditions, even if application messages are used instead of heartbeats. By this means, the failure detector is able to adapt even faster to changing networking conditions, as application and heartbeat messages can be used to sample inter-arrival times. Due to the fact that every message is used to draw conclusions about the network's condition, a much richer set of training information is available. To realise the lazy heartbeat sampling, q is supposed to append a consecutively numbered id and the sending time according to its local clock to every message it sends to q. It is shown in the following that, with this additional information, process p can use every message it receives to compute a sample s for the sample window S.

While heartbeat messages are sent every Δ_i, application messages can be sent at random times. The following Formula 2.5 transforms the inter-arrival times of the application messages into inter-arrival times representing an estimation for heartbeats sent instead of the application messages.

In this way, application messages are made useful for the failure detection task. This formula is the core of the lazy monitoring and consists of three parts:

$$s = \underbrace{\Delta_i}_{\textit{heartbeat interval}} + \underbrace{(l \cdot \Delta_i)}_{\textit{message loss}} + \underbrace{(\Delta_r - \Delta_s)}_{\textit{sending time}} \tag{2.5}$$

Heartbeat interval: The heartbeat interval Δ_i is the expected value for the inter-arrival times of heartbeats and the basis for the lazy heartbeat inter-arrival time estimation.

Message loss: Lost messages can be identified by the piggy-backed message-ids. Every consecutively lost message lengthens the estimated heartbeat inter-arrival time by Δ_i. l is the number of consecutively lost messages which arises from the message-ids (l = current received message-id - last received message-id -1). Therefore $l \cdot \Delta_i$ adds the additional time by lost messages to Formula 2.5.

Sending time: The term $\Delta_r - \Delta_s$ reflects the variations of the inter-arrival time through the network. Δ_r is the difference of the receipt times of the current message and the previously received message according to p's local clock and Δ_s is the difference of the sending times of the current message and the previously received message according to q's local clock which can be computed by p with the appended sending times.

Algorithm 6 presents the lazy heartbeat sampling in detail.

To clarify the functionality of the new lazy monitoring approach two examples are discussed.

Example 1 In this first example it is assumed that two application messages are sent within an interval that is shorter than the heartbeat interval. None of these two messages gets lost.

Δ_i**:** heartbeat interval is 1000 ms

m_{i-1}**:**
- message-id: i-1
- sending time: 0000 (according to q's clock)
- receipt time: 0500 (according to p's clock)

m_i**:**
- message-id: i
- sending time: 0250 (according to q's clock)
- receipt time: 0800 (according to p's clock)

The messages m_{i-1} and m_i are two application messages that are sent from q to p. The message-ids i and $i-1$ indicate that no message has been lost and thus $l = 0$ in this case. The message m_i has been sent exactly 250 ms after

Algorithm 6 Lazy heartbeat sampling

```
1:  Process q:
2:      append sending time t_s and consecutive id to every outgoing mes-
    sage
3:
4:      if no message sent to p since Δ_i then
5:          send heartbeat message to p
6:      end if
7:
8:  Process p:
9:
10:     f = −1                                          ▷ freshness point
11:     S = nil                                ▷ S is initialised as an empty list
12:
13:     procedure RCV_M(m_j, t_r)              ▷ receiving message m_j at time t_r
14:         if f = −1 then
15:             f = t_r
16:         else
17:             s = Δ_i + (l · Δ_i) + (Δ_r − Δ_s)
18:             f = t_r
19:             append s to S
20:         end if
21:     end procedure
22:
```

m_{i-1}. The interesting point here is that the sending times have an interval of 250 ms, the receipt times of 300 ms seconds. Thus the sending time variation is 50 ms.

The basic idea of this lazy monitoring approach is roughly spoken "if the application message were a heartbeat what would a sample look like". If the values of this example are inserted into Equation 2.5, the result is:

$s = \Delta_i + (l \cdot \Delta_i) + (\Delta_r - \Delta_s) = 1000\,ms + (0 \cdot 1000\,ms) + (300\,ms - 250\,ms) = 1050\,ms$

The inter-arrival time of m_i and m_{i-1} is 300 ms - the sending interval is 250 ms. Heartbeat messages are supposed to be sent every 1000 ms. Thus if these two application messages were heartbeat messages then they would have been sent with an interval of 1000 ms and the inter-arrival time that q had experienced would be 1050 ms.

Example 2 In the second example a case is analysed where one message gets lost. This can be recognised with the message-ids.

Δ_i**:** heartbeat interval is 1000 ms

m_{i-2}:
- message-id: i-2
- sending time: 0000 (according to q's clock)
- receipt time: 0300 (according to p's clock)

m_i:
- message-id: i
- sending time: 0800 (according to q's clock)
- receipt time: 1030 (according to p's clock)

With this setting the sample s is calculated as follows:

$s = \Delta_i + (l \cdot \Delta_i) + (\Delta_r - \Delta_s) = 1000\,ms + (1 \cdot 1000\,ms) + (730\,ms - 800\,ms) = 1930\,ms$

In this example p sends three application messages to q and the second gets lost. With the same probability the loss could also have happened to a sent heartbeat message. Furthermore the sending time of m_i is 70 ms shorter than the sending time of m_{i-2}. This example transferred to the case of real heartbeat messages would result in an inter-arrival time of 1930 ms. The meaning of Equation 2.5 should be clearer now: every message is used to draw conclusions about the conditions a heartbeat would undergo if sent at the same time. Process q passes on sending real heartbeat messages if application messages are sent. Process p, however, is still able to sample heartbeat inter-arrival times and can update its knowledge base, the list of inter-arrival times S. The failure estimation algorithm of heartbeat-style failure detectors do not have to be changed in order to apply this lazy monitoring approach. If processes are communicating frequently no single heartbeat message has to be sent and nearly no network overhead is produced by failure detectors.

It is also possible to improve the quality of failure detectors by reducing the heartbeat interval Δ_i. Thus failures can be detected faster - the bigger network overhead caused by this action could cease to exist with the application of the lazy monitoring approach. If no application messages are sent then of course heartbeat messages have to be used. But it can be argued that in this case there is a very low network load and the overhead caused by heartbeat messages can be borne. However, during high traffic times where additional traffic would be very unpleasant the proposed approach prevents the network from this overhead.

It has to be mentioned that the lazy monitoring approach introduces overhead as a message id and the sending time is appended to application messages. However, this overhead can be reduced to 4 bytes or even less per message. Suppose the message id is represented with 10 bits and the timestamp with 22 bits. Then, the message id has a range from 1 to 1024, and a timestamp can be represented by the number of milliseconds since the last full hour. The check for lost messages as well as the delay calculation has then to be performed with modulo 2^{10} and 2^{22} respectively. Of course p

is now unable to distinct e.g. whether none or 1024 consecutive messages have been lost, but the latter case is very unlikely and can be ignored. The analogue is holding for the appended timestamp.

In Section 2.5.3 a different strategy for setting the freshness points is presented. This technique applied to the lazy monitoring results in a slightly changed equation for taking lazy monitoring samples:

$$s = \Delta_i + (l \cdot \Delta_i) + (t_r - t_s) \tag{2.6}$$

where

- t_r is the receipt time of the message according to p's clock and
- t_s is the sending time of the message according to q's clock.

Besides the usage of Formula 2.6 instead of 2.5 all lazy monitoring concepts can be applied without change to a failure detector using the different freshness point strategy.

2.7.2 Message selection strategy

In this section two issues are addressed. First, until now it is assumed that the behaviour of application messages is a good estimator for the behaviour of heartbeat messages. However, this is not the case if application messages can become much larger than heartbeats. One way to overcome this is to perform the piggybacking of the relevant information at packet level. But as the engineering of this heavily depends on the used protocols and often the access to these layers is neither given nor wanted, lazy monitoring is only considered on the application/message level in this work. Second, up to now lazy monitoring information is appended to all application messages - this is not always necessary.

A message selection strategy determines which messages to use for lazy failure detection. This is useful to (1) omit application messages which are too large and therefore unsuitable as information source for failure detectors and (2) to adjust the amount of suitable messages which are used for lazy monitoring. A message selection strategy (MSS) takes as input an application message and outputs whether it is used for lazy monitoring, i.e. whether a message id and a timestamp are appended. The simplest strategy is to omit all application messages which are larger than $MAXSIZE$ bytes and to use the remaining ones. Application messages smaller or equal to $MAXSIZE$ are supposed to have similar behaviour to heartbeat messages and therefore considered suitable for lazy monitoring. This MSS is illustrated in Algorithm 7.

Algorithm 7 Simple MSS

```
1:  Process q:
2:
3:      procedure MSS(m)                ▷ Called for every outgoing message m
4:          if size of m ≤ MAXSIZE then
5:              append sending time t_r and consecutive id to m
6:          end if
7:      end procedure
8:
9:      if no selected message sent to p since Δ_i then
10:         send heartbeat to p
11:     end if
12:
```

If all suitable application messages are used for lazy monitoring a maximum number of samples is provided to the failure detector. As some failure detectors might not be able to profit from an amount of samples higher than a certain value n_u per interval Δ_i even suitable application messages can be omitted to reduce overhead. Suppose n_s is the average number of suitable messages within an interval Δ_i and n_u is the maximum number of messages the failure detector can utilise as samples. The adaptive MSS of Algorithm 8 reduces the number of messages used for lazy monitoring to n_u on average.

Algorithm 8 Adaptive MSS

```
1:  Process q:
2:
3:      procedure MSS(m)                ▷ Called for every outgoing message m
4:          if size of m ≤ MAXSIZE and rand() < n_u/n_s then
5:              append sending time t_r and consecutive id to m
6:          end if
7:      end procedure
8:
9:      if no selected message sent to p since Δ_i then
10:         send heartbeat to p
11:     end if
12:
```

The function $rand()$ returns a random value within the interval $[0, 1)$.

	non-lazy	lazy
(1) Traffic	h bytes	$b \cdot n_{MSS} + p \cdot h$ bytes
(2) #Messages	1	$p \qquad (0 \leq p \leq 1)$
(3) #Samples	1	$max(1, n_{MSS})$

Table 2.5: *Comparison non-lazy/lazy failure detection*

2.7.3 Evaluation

In this section the costs and benefits of the lazy monitoring approach is described with respect to (1) the produced traffic, (2) the number of sent messages, and the (3) number of samples provided to the failure detector. The points (1) and (2) should be as small as possible, but for (3) higher values are better because then the failure detector is provided with more information about the environment. To make statements about these three measures, the following variables are used:

- Δ_i: the heartbeat interval,
- h: the size of a heartbeat message (including header),
- b: the size of data needed to be appended in order to use an application message as sample,
- n_{MSS}: the average number of messages the MSS selects within Δ_i, and
- p: the probability that no message is selected within Δ_i.

Table 2.5 shows how to compute (1), (2), and (3) where all measures refer to the interval Δ_i.

The introduced lazy monitoring technique is applied within the Smart Doorplate Project [TBPU03a, TBPU05]. This project envisions the use of smart doorplates within an office building. The doorplates are amongst others able to display current situational information about the office owner and to direct visitors to his current location based on a location-tracking system. A middleware called "Organic Computing Middleware for Ubiquitous Environments" OCμ [Tru06] based on JAVA and JXTA serves as common platform for all included devices. To detect failures, the devices monitor each other using failure detectors. The lazy monitoring is used to minimise the messaging overhead caused by these failure detectors.

In the smart doorplate environment the heartbeat interval of the failure detectors Δ_i is set to 10 seconds. Messages are exchanged in XML format in JXTA leading to an overhead of 3.5 kB per message. This overhead does not only consume network resources but also represents a computational overhead for each node sending and receiving messages. The amount of data which has to be appended to an application message in order to use it for lazy monitoring are in the test case 4 bytes. Especially the location tracking-

	non-lazy	**lazy**
(1) Traffic	3500 bytes	43.5 bytes
(2) #Messages	1	0.01
(3) #Samples	1	10

Table 2.6: *Comparison non-lazy/lazy failure detection within the Smart Doorplate Project*

system typically generates many small messages containing coordinates of the office owners which all can be used for the lazy monitoring. The amount of selected messages is limited to 10 per 10 seconds, using the adaptive MSS shown in Algorithm 8. The probability that no single message is sent during Δ_i (10s) depends strongly on the system, the number and types of services communicating in the network, the daytime, and so on. In the test case a value of 1% is even pessimistic. This leads to the following list:

- Δ_i: 10 seconds,
- h: 3.5 kB,
- b: 4 Bytes,
- n_{MSS}: 10, and
- p: 1%.

Table 2.6 summarises and compares the resulting traffic, the number of messages sent by the failure detector, and the number of samples the failure detector can use to adapt to the network.

These results show that by using the lazy monitoring approach the generated traffic can be reduced significantly as well as the number of sent messages. Furthermore, more samples are provided to the failure detector and this allows it to adapt faster to changing network conditions. In the testbed the usage of lazy monitoring reduces the traffic to 1.2% and the number of messages to 1% of the benchmark while 10 times more information about the environment is available.

2.7.4 Processing delays

The lazy monitoring introduced so far only considers network delays. It is based on the assumption that a process is able to send heartbeats exactly with a frequency of one heartbeat per Δ_i seconds. To detect the processing delays the regular inter-arrival times of heartbeats can be compared with samples estimated by Formula 2.5.

Consider the following example where q is sending a heartbeat message. If q sends the heartbeat exactly after Δ_i like it is obligated, then the lazy sampling and the non-lazy sampling always produce the same values:

Δ_i**:** heartbeat interval is $1000ms$

m_{i-1}**:**
- message-id: i-1
- sending time: 0000 (according to q's clock)
- receipt time: 0050 (according to p's clock)

m_i**:**
- message-id: i
- sending time: 1000 (according to q's clock)
- receipt time: 1100 (according to p's clock)

If these values are inserted into equation 2.5 the result is

$s = \Delta_i + (l \cdot \Delta_i) + (\Delta_r - \Delta_s) = 1000ms + (0 \cdot 1000ms) + (1050ms - 1000ms) = 1050ms$,

which corresponds to the real inter-arrival time. Now an example is presented where q is unable to send a heartbeat accurately timed:

Δ_i**:** heartbeat interval is $1000ms$

m_{i-1}**:**
- message-id: i-1
- sending time: 0000 (according to q's clock)
- receipt time: 0050 (according to p's clock)

m_i**:**
- message-id: i
- sending time: 1080 (according to q's clock)
- receipt time: 1100 (according to p's clock)

The real inter-arrival time of the heartbeats is again $1050ms$. The formula, however, computes the following:

$s = \Delta_i + (l \cdot \Delta_i) + (\Delta_r - \Delta_s) = 1000ms + (0 \cdot 1000ms) + (1050ms - 1080ms) = 970ms$.

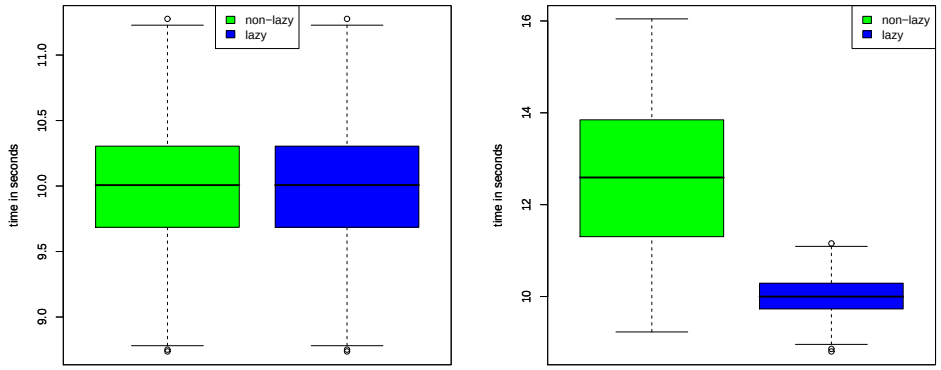
The lazy heartbeat sampling introduced so far results in correct values if q is able to send heartbeats at the right time. If the processing delays of q have considerable influence on the inter-arrival times the lazy sampled values could be falsified.

To further illustrate the effects of the process imprecision concerning sending heartbeats, the following example is considered: Heartbeat interval Δ_i is $10s$. The time it takes to transmit a message is assumed to be normal distributed $\mathcal{N}(1s, 0.3s)$ with mean $1s$ and standard deviation $0.3s$. The normal distribution is an often used instrument to model message delay. For the following examples this is not a crucial point and other distributions could be used.

Figure 2.53(a) shows a Box-Whisker-Plot[2] of two sampling windows, one

[2] A Box-Whisker-Plot is a diagram to graphically represent numerical data. The lines of the

taken with one without lazy sampling, under the assumption that q sends heartbeats accurately timed.



(a) No process imprecision (b) High process imprecision of 5s

Figure 2.53: *Comparison lazy/non-lazy sampling*

Figure 2.53(b) shows a similar Box-Whisker-Plot, but now q sends heartbeats at a random time within $[\Delta_i, \Delta_i + 5s]$ and not at Δ_i as in Figure 2.53(a). You can see that now the two datasets differ significantly. Caused by the imprecision of q, the lazy heartbeat sampling results in falsified values as it disregards this imprecision.

What are the effects of such a process imprecision on the failure detection using lazy monitoring? To compute a failure probability the introduced failure detection algorithm uses the cumulative frequencies of the entries in the sampling window. Figure 2.54 illustrates the cumulative probabilities of the above introduced example involving the highly inaccurately timed behaviour as shown in Figure 2.53(b). Obviously, the cumulative frequencies of the lazy sampled values differ significantly from the ones sampled traditionally.

To overcome this issue, in the following an *adaptive* lazy heartbeat sampling algorithm is presented. Whenever q sends a real heartbeat, p draws conclusions about q's imprecision and adjusts the lazy sampling according to this information. The adaptive lazy heartbeat sampling is shown in Algorithm 9.

Whenever a heartbeat message is received the imprecision can be evaluated.

box represent the lower quartile, median, and upper quartile values. Whiskers extend from each end of the box to the most extreme values within 1.5 times the interquartile range from the ends of the box. Outliers beyond the ends of the whiskers are depicted as circles. For more detailed information it is referred to [Tuk77, R D05]

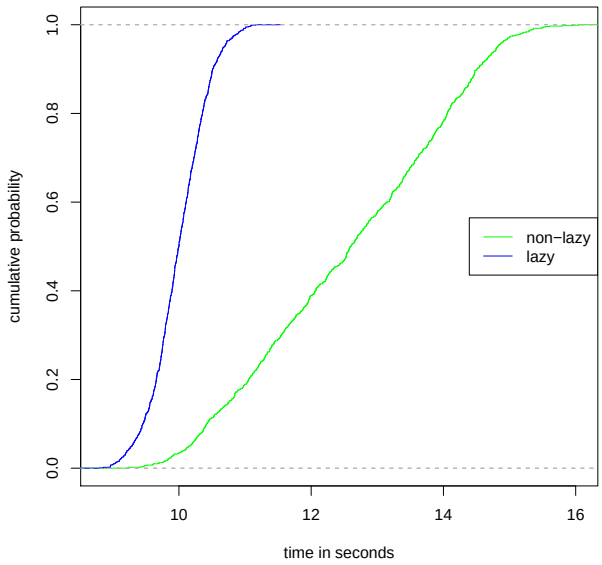


Figure 2.54: *Effect of process imprecision on failure detection*

If q is sending the heartbeat accurately timed, then s_c and s_l are equal and s_Δ is 0. If the heartbeat is not sent at the right time, then s_Δ presents the imprecision of process q sending this heartbeat. The values of s_Δ are stored in a list I. When p is receiving an application message it uses the imprecision values in I to adjust the lazy sampled values.

In the following the adaptive lazy heartbeat sampling is experimentally evaluated with this setting:

A process q is monitored by p. The heartbeat interval is Δ_i. In the case of non-lazy monitoring no application messages are used to sample inter-arrival times. In this case the sampled values are indeed correct but q has to send a heartbeat message to p every Δ_i seconds. Heartbeat messages cannot be saved using application messages. Using lazy heartbeat-sampling, no single heartbeat message is needed if application messages are sent frequently. But if q is working imprecisely regarding timing requirements the lazy sampled values are falsified. To overcome this falsifications, adaptive lazy monitoring uses heartbeats to adjust the sampling to the process' timing imprecision. In the conducted experiments it has been assumed that on an average every tenth message is a heartbeat message that can be used to adjust the adaptive lazy sampling. The experiments differ in the imprecision of q and the size η of the sampling window S. In the first four experiments of Figure 2.55 a sampling window size η of 1000 has been used. In the last four experiments (see Figure 2.56 - 2.56) a sampling window size η of 20000 has been used. The imprecision of q has been modelled in delaying the sending of heartbeats for a random time within the interval $[0, p_i]$.

Algorithm 9 Adaptive lazy heartbeat sampling

Process q:
    append sending time t_s and consecutive id to every outgoing message

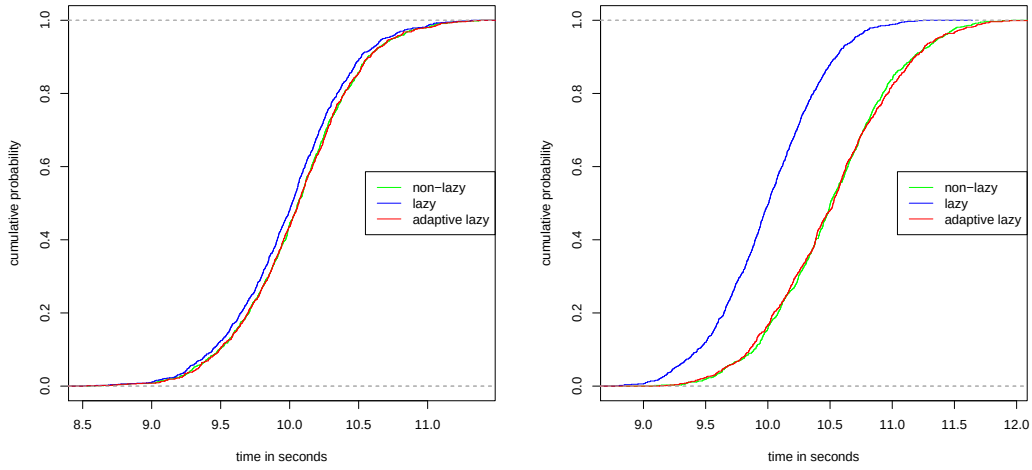
    if no message sent to p since Δ_i **then**
        send heartbeat message to p
    end if

Process p:

    $f = -1$ ▷ freshness point
    $S = nil$ ▷ S is initialised as an empty list
    $I = nil$ ▷ I contains data about q's imprecision

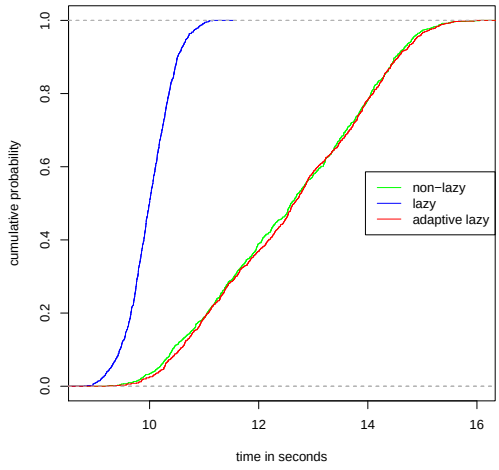
    procedure RCV_M(m_j, t_r) ▷ receiving message m_j at time t_r
        if $f = -1$ **then**
            $f = t_r$
        else
            if m_j is heartbeat **then**
                $s_c = t_r - f$ ▷ Conventionally sampled inter-arrival time
                $s_l = \Delta_i + (l \cdot \Delta_i) + (\Delta_r - \Delta_s)$ ▷ Lazy sampled inter-arrival time
                $s_\Delta = s_c - s_l$ ▷ Difference between s_c and s_l
                append s_Δ to I
                append s_c to S
            else
                pick a random value i_r from I
                $s_{al} = s_l + i_r = \Delta_i + (l \cdot \Delta_i) + (\Delta_r - \Delta_s) + i_r$
                append s_{al} to S
            end if
            $f = t_r$
        end if
    end procedure

Thus, higher values for p_i represent a process with a lower ability to send heartbeats accurately timed. The values that have been used for p_i are 0.1s, 1s, 5s, 10s.

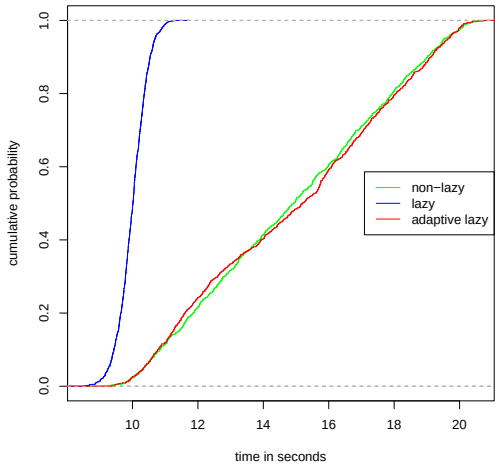


(a) p_i: 0.1s

(b) p_i: 1s

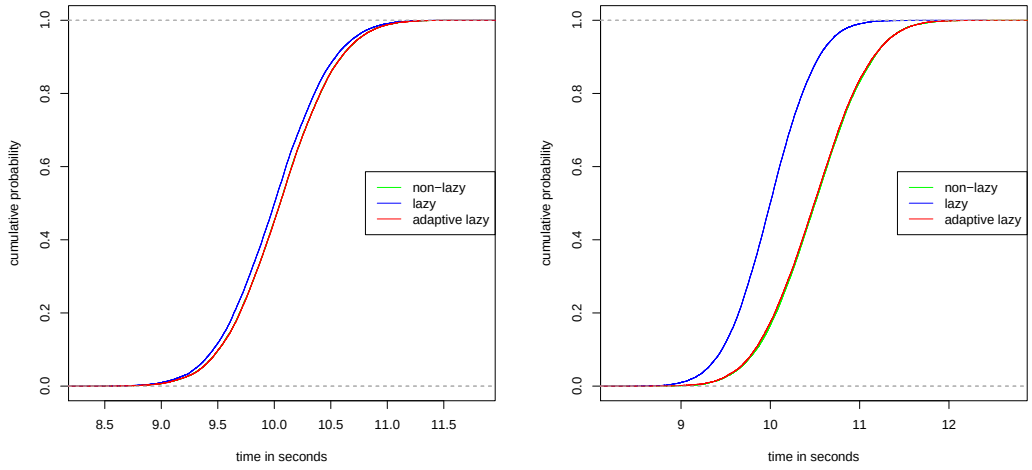


(c) p_i: 5s

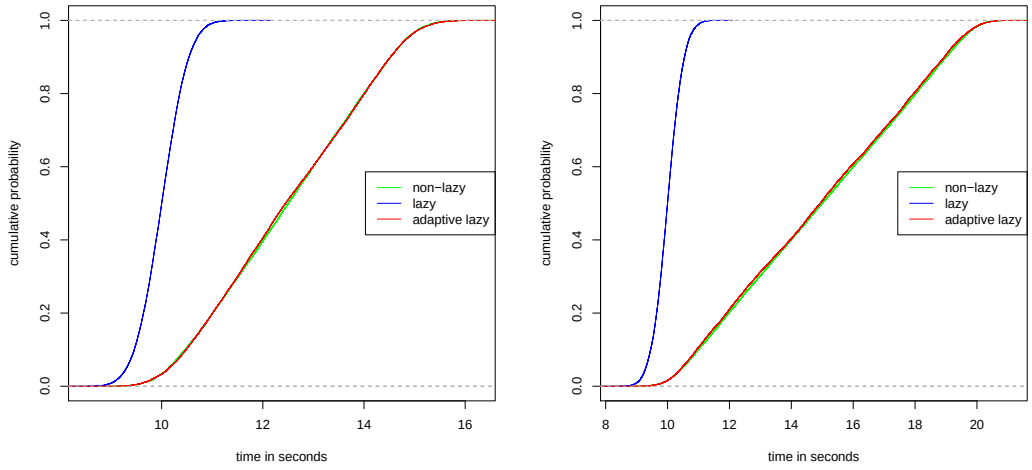


(d) p_i: 10s

Figure 2.55: *Adaptive lazy monitoring results, sampling window size:* 1000



(a) p_i: 0.1s (b) p_i: 1s



(c) p_i: 5s (d) p_i: 10s

Figure 2.56: *Adaptive lazy monitoring results, sampling window size:* 20000

Figure 2.55 and 2.56 shows that depending on the imprecision of the process q, the non-adaptive lazy inter-arrival time sampling differs from the real non-lazy sampled values. Using the adaptive lazy sampling, however, the sampled values do not differ significantly from the conventionally sampled ones. Even for an extremely unreliable process that sends messages anytime within a ten second delay interval the adaptive lazy sampling shows very good results. Using a sampling window of size 1000 a slight but very small difference between the conventionally and adaptive lazy sampled values is observable. Larger sized sampling windows lead to a stabilisation of the conventional and adaptive lazy sampled values.

2.8 Conclusions

In this chapter, a new failure detection algorithm has been presented. It is an adaptive accrual failure detector, applicable to a wide area of scenarios, and adequate to build generic failure detection services. Despite algorithms that depend on distribution models for heartbeat inter-arrivals like the Gaussian distribution, it builds its own model based on the observed values. This avoids the problem of model inadequacy and increases the versatility. Furthermore, it makes very low demands on the computational power of the hardware, comes with concepts to reduce network load, is mathematically founded, and very insusceptible to message loss. All these features affirm that this failure detector could be very interesting also for environments where failure detection is problematic, such as in ubiquitous systems.

Performance measurements have been conducted to compare the new failure detector to the well-known failure detectors of Chen et al. [CTA00] and Bertier et al. [BMS02], and the accrual failure detector of Hayashibara et al. [HDYK04]. The detailed measurements show that the performance of the failure detection algorithm in the experiments is excellent and outperforms the other algorithms.

Additionally, a set of variations and modifications of the new failure detector has been introduced and evaluated. Most of these variations could help to even improve the performance of the basic version of the failure detector. A variation of the algorithm proposed in this work makes about 90% less wrong suspicions than other state of the art algorithms. Some of the presented concepts the variations are based on could also be interesting for other failure detection algorithms to improve their performance.

To conduct performance measurements of different failure detection algorithms with minimal effort, a test centre for failure detectors has been developed. This framework is programmed in JAVA and provides all tools and concepts to evaluate a failure detection algorithm - the only task that

has to be done is to specify the failure detection algorithm by implementing two methods of an abstract class.

Furthermore, a mechanism called lazy monitoring is presented with the ability to significantly reduce the overhead caused by heartbeat-style failure detectors. These do not need to be changed in order to apply this approach. Besides the reduction of overhead, the lazy monitoring also contains the possibility of a faster adaptation to changing network conditions and better detection quality due to more information about the network. In the conducted evaluations the usage of lazy monitoring could reduce the traffic to 1.2% and the number of messages to 1% while 10 times more information about the environment is available. The saved resources can be utilised to enable a faster failure detection. Especially for environments with battery-powered devices, like sensors in smart environments or sensor networks, the reduction of message sending is very valuable as each sent message consumes a relatively high amount of power. The presented techniques could be a key enabler for using failure detectors in such environments.

3

Monitoring groups

3.1 Introduction

In the last chapter, efficient techniques that allow a node to monitor another node are presented. To enable self-healing distributed systems, an important ingredient is a scalable *self-monitoring* capability. An autonomous formation of monitoring relations is needed to extend the failure detection techniques to that effect. A simple technique would be that any two nodes in a distributed system monitor each other. As this results in a huge overhead and unscalable behaviour, intelligent strategies are needed to manage the monitoring responsibilities.

This chapter proposes algorithms to autonomously install monitoring relations in distributed systems. These techniques are tailored to work in complex, large scale, distributed systems allowing a fast formation of monitoring groups to minimise the time that nodes are unmonitored.

In the next section, a survey of related work is given. Section 3.3 describes the contributions of the approach proposed in this chapter, and Section 3.4 gives a precise problem statement for the establishment of monitoring groups. Then, in Section 3.5, three grouping algorithms are introduced and evaluated in Section 3.6. Finally, Section 3.7 concludes this chapter.

3.2 Related work

Scalable failure detectors

To supply adequate support for large scale systems, *hierarchical* failure detectors define some hierarchical organisation. Bertier et al. [BMS03] introduce a hierarchy with two levels: a local and a global one, based on the underlying network topology. The local groups are LANs, bound together by a global group. Every local group elects one leader that is member in the global group. Within each group any member monitors all other members.

Figure 3.1 illustrates such a hierarchy with two groups at level 1 and one group at level 2. Referring to Bertier's taxonomy the groups at level 1 are the local groups, and the group at level 2 is the global group. Different from Bertier's failure detector, organisations with higher group levels are also possible.

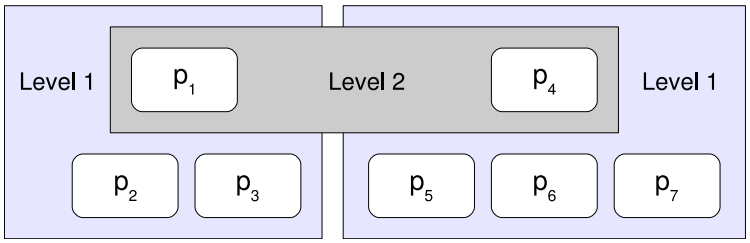


Figure 3.1: *Hierarchical failure detectors*

Hierarchical failure detectors demand a differentiation of the semantics of failures on different levels. Failures on the lowest level normally correspond to failures of a single process, whereas the detection of a failure on a higher level indicates the crash of the entire subgroup. Different from Bertier et al. [BMS03], in this work the existence of some classifying concept like a LAN is not required. Monitoring groups can also be built within a network of equal nodes. A hierarchy as proposed in [BMS03] is not further investigated in this work, but could be easily built upon the *monitoring groups* which are introduced later on.

Another hierarchical failure detector is presented by Felber et al. [FDGO99] proposing an approach to view failure detectors as first class objects. They emphasise the importance of well defined interfaces for failure detectors in order to e.g. reuse existing failure detectors. In their paper they also present a hierarchical configuration of the monitoring system to raise efficiency and scalability.

Gossipping is a method of information dissemination within a distributed system by information exchange with randomly chosen communication

partners. In 1972, Baker and Shostak [BS72] discussed a gossipping system with ladies and telephones. They investigated the problem of n ladies, each of them knows some item of gossip not known to the others. They use telephones to communicate, and the ladies tell everything they know at that time whenever one lady calls another. The problem statement was "How many calls are required before each lady knows everything?". Demers et al. [DGH+87] pioneered gossipping in computer science as a way to update and ensure consistent replicas for distributed databases.

Van Renesse et al. [RMH98] have been the first using gossipping for failure detection to cope with the problem of scalability. In their basic algorithm each process maintains a list with a heartbeat counter for each known process. At certain intervals every process increments its own counter and selects a random process to send its list to. Upon receipt of a gossip message the received list is merged with its own list. Each process also maintains the last time the heartbeat counter has increased for any node. If this counter is increased for a certain time then the process is considered to have failed. Additionally to this basic gossipping, the authors specify a multi-level gossipping algorithm that does not choose the communication partners completely randomly but dependent on the underlying network. Basically, they try to concentrate the traffic within subnets and to decrease it across them to further improve scalability. A disadvantage is that the size of gossip messages grows with the size of processes, and this causes a relatively high network traffic. Furthermore, the timeout to prevent false detections has to be rather high and since every process checks failures of processes by its own, false detections cause inconsistent information.

The SWIM protocol, based on the work of Gupta et al. [GCG01] and described in a paper of Das et al. [DGM02], uses a separate failure detection and failure dissemination component. The former detects failures while the latter distributes information about processes that have recently either left, joined, or failed. Each process periodically sends a ping message to some randomly chosen process and waits for it to respond. In this way failures can be detected and are then disseminated by a separate gossip protocol. The separation of failure detection and further components as proposed in [DGM02] is taken up in this work. While the previous chapter introduces the failure detection component, here the dissemination component is proposed.

Horita et al. [HTC05] present a scalable failure detector that creates dispersed monitoring relations among participating processes. Each process is intended to be monitored by a small number of other processes. In almost the same manner as in systems mentioned above, a separate failure detection and information propagation is used. Their protocol tries to maintain each process being monitored by k other processes. As a typical number for k they declare 4 or 5. When a process crashes, one of the monitoring

processes will detect the failure and propagate this information across the whole system. In addition to the description of their failure detector, Horita et al. compare the overheads of different failure detection organisations in their paper. The grouping mechanism of Horita et al. [HTC05] is based on a random construction of monitoring relations. Each node selects a certain amount of randomly chosen nodes which then serve as its surveillants. Hence, it is not taken into account how well a node is suited to monitor another. One motivation for the concepts proposed in this chapter is to take such an optimality criterion into account.

Graph partitioning

Graph partitioning represents a fundamental problem arising in many scientific and technical areas. In particular, understanding the graph as a network, it is a problem closely related to the problem approached in this work. Consider each partition of a network as a group of nodes which monitor each other.

A *k-way partition* of a weighted graph is the partitioning of the node set into k disjoint subsets, so as to minimise the weight of edges connecting nodes in different partitions. This problem is known to be NP-hard [GJ90] while many heuristics and approximation algorithms are known which aim at producing solutions close to the optimum. However, most of these techniques are not applicable to distributed environments and are therefore unsuitable to form monitoring groups.

An algorithm capable of solving a slightly modified k-way partition problem in a distributed way is presented by Roy et al. [RWS06]. It is based on a stochastic automaton called influence model [ARLV01]. An influence model consists of a network of nodes which can take one of a finite number of statuses at discrete time steps. At each time step the algorithm proposed in [RWS06] performs the following: Each node picks a node as determining node with a certain probability and copies its status. By recursively performing these steps, partitions emerge. They argue that, under some constraints, their algorithm finds partitions which pass to the optimal partition with probability 1.

In this work two algorithms to partition a network into groups are introduced. This is a problem very similar to graph partitioning. However, the problem investigated in the following is adapted to the needs of self-healing distributed systems.

3.3 Contribution

The contribution of this chapter is the introduction and evaluation of algorithms to form monitoring relations and monitoring groups respectively. The grouping component is independent from the used monitoring component. The latter could for instance be a failure detector as introduced in Chapter 2 or any other mutual monitoring task. The separation of the monitoring itself and the group formation allows to create generic monitoring and grouping services. As clarified in the previous section, the separation of information propagation and monitoring has been identified as an important characteristic by many researchers.

In the area of scalable failure detectors, the consideration of the suitability of monitoring relations has been neglected so far. For instance, the work of Horita et al. [HTC05] proposes to choose surveillants *randomly*. Taking suitability information into account can improve the performance and reduce the overhead of monitoring components like failure detectors. Related methods from graph partitioning, which in fact search for optimal relations, are too complex and slow for an application in complex systems. Furthermore, graph partitioning algorithms normally need global knowledge and are not designed to work in a distributed environment. For self-healing systems a fast installation of monitoring relations is more important than to find an optimal solution eventually. Especially from the point of view that a network can be subject to changes what means an optimal solution could become obsolete faster than it is found.

To cover a wide range of different requirements and applications, dispersed monitoring relations, as arising if each node chooses its surveillants individually [HTC05], as well as closed monitoring groups which result from e.g. network partitioning, are studied.

In the following, a precise formal definition of the stated problem is given.

3.4 Problem statement

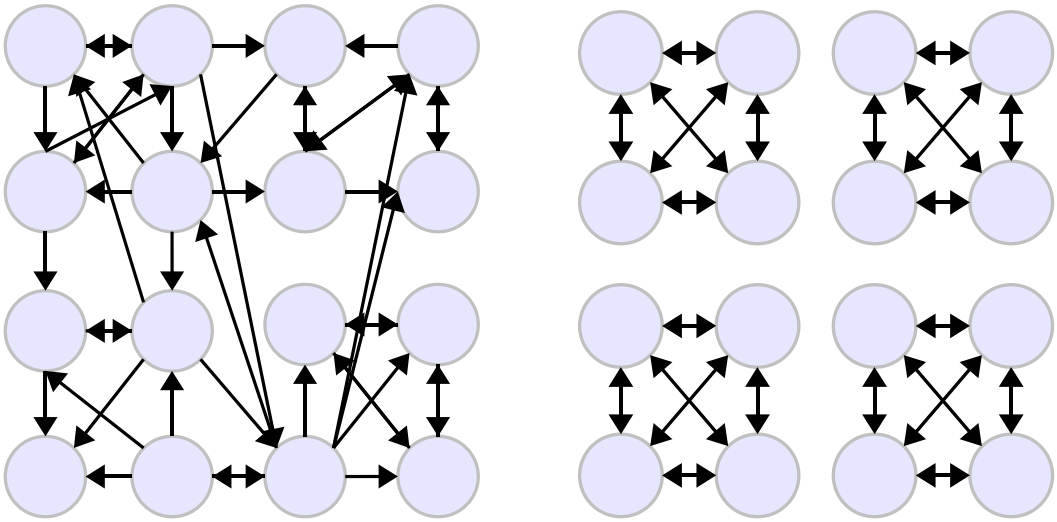
A *monitoring network Net*, a network of monitoring relations, is represented as a triple (N, M, s), where N is the set of nodes/processes of a network, $M \subseteq N \times N$ is the monitoring relation, and s is a function from $N \times N$ to a real value within $[0, 1]$. For each tuple $(u, v) \in N \times N$, $s(u, v)$ is the suitability of node u to monitor node v. This suitability can depend on different aspects like the latency of a connection, the reliability of a node, its load and so on. If a node u is not able to monitor another node v at all, $s(u, v)$ should output 0. The monitoring relation defines which monitoring relations are

established, i.e. $(u, v) \in M$ means node u is currently monitoring node v. $(u, v) \in M$ is also denoted with $u \rightarrow v$. The relation M is irreflexive, i.e. it is not allowed that a node is monitoring itself. The term $\overset{*}{\rightarrow} v$ is defined as all nodes monitoring v, i.e. $\overset{*}{\rightarrow} v := \{u \in N \mid u \rightarrow v\}$. Similar, $u \overset{*}{\rightarrow}$ outputs all nodes u is monitoring, i.e. $u \overset{*}{\rightarrow} := \{v \in N \mid u \rightarrow v\}$.

The task of a grouping algorithm is basically, given a monitoring network $Net = (N, M, s)$ and a positive integer $m < |N|$, to establish monitoring relations such that every node of the network is monitored by at least m nodes. In this work two flavours of this problem are distinguished, namely *individual monitoring relations* also called *dispersed monitoring relations* and *closed monitoring groups*. In the former, monitoring relations can be set for each node individually while in the latter nodes form groups with mutual monitoring relations.

The number m of surveillants for each node can be defined by the user. Typically, a higher number of surveillants provides a higher reliability but also causes a higher overhead.

In Figure 3.2(a), an instance of individual monitoring relations of a monitoring network is illustrated with $m = 3$. Thereby, the illustration of the suitability information has been omitted. Figure 3.2(b) shows a corresponding partition of a network into monitoring groups.



(a) Individual monitoring relations (b) Monitoring groups

Figure 3.2: *Types of monitoring relations*

In the following, problem definitions of establishing individual monitoring relations and monitoring groups are given.

Individual monitoring relations

Given a positive integer m, where $m < |N|$, establish monitoring relations M where $\forall n \in N$ holds $|\xrightarrow{*} n| = m$. This means each node is monitored by m other nodes. Furthermore, the algorithm should maximise the suitability of the grouping to establish adequate monitoring relations. Therefore the term

$$\sum_{v \in N} \sum_{u \in \xrightarrow{*} v} s(u, v)$$

should be maximised by the grouping algorithm. The optimisation of the suitability is a quality criterion for grouping algorithms, but it is not postulated that the algorithms output an optimal solution as it is more important to find solutions in all cases as fast as possible.

The term *monitoring group* or simply *group* in the context of individual monitoring relations can be understood as all nodes monitoring one particular node, and the latter is the leader of the group. Thus, in a network of n nodes the are also n groups: Each node $v \in N$ is the leader of the group $\{v\} \cup \xrightarrow{*} v$.

Closed monitoring groups

Different from the dispersed individual monitoring relations, a closed monitoring group is a group of nodes where all members monitor each other. This problem is very similar to a graph partitioning problem. In addition to the individual monitoring relations, constraints regarding the monitoring relations M are holding: M must be symmetric and transitive in order to produce closed monitoring groups. In another point, the problem of finding monitoring groups is relaxed, compared to individual monitoring relations, as it is not always possible to find groups of the size $m + 1$ resulting in m surveillants per node in the group. If for instance a network has three nodes and monitoring groups of size 2 need to be established, this leads to an unsolvable problem. For such cases, also closed monitoring groups of bigger sizes are allowed. In detail, the problem $\forall n \in N$ holds $|\xrightarrow{*} n| = m$ is relaxed to $\forall n \in N$ holds $|\xrightarrow{*} n| \geq m$. A very simple solution to this problem is to combine the whole network into one group. This is a valid solution as just $|\xrightarrow{*} n| \geq m$ is postulated. However, the number of surveillants per node should be as close as possible to m. This represents a soft constraint similar to the maximisation of the suitability criterion.

Two nodes are in the same closed monitoring group if they are monitoring each other. An additional requirement for such monitoring groups is that each group has one node which is declared as leader. Such a role is needed

by many possible applications based upon grouped nodes, e.g. to have one coordinator or contact for each group. An instance where one leader per group is necessary is the formation of hierarchical groups.

Whether individual monitoring relations or monitoring groups are more adequate depends on the environment and the monitoring task. Furthermore, the installed groups can also be used for many other purposes beyond monitoring, such as cooperative failure recovery.

3.5 Grouping algorithms

In this section three grouping algorithms are introduced, one to establish individual monitoring relations, two to form closed monitoring groups. The algorithms are tailored to solve these problems in a distributed manner. Furthermore, it is not assumed that all nodes have information about all other nodes what would simplify the problem significantly. The nodes of a monitoring network $Net = (N, M, s)$ do not know about the suitability s, i.e. how suitable other nodes are to monitor it, until they receive a message from a node with information about that. The suitability also might change over time. In the following, the usage and relevance of suitability metrics for monitoring relations is discussed. Then, three algorithms are presented which provide the desired grouping capabilities.

In order to establish suitable monitoring relations, the nodes of a network need information about each other. Such information might be the quality of the network connection of two nodes, the reliability of a node, and so on. Each node is holding relevant information about a number of other nodes allowing to compute suitability information.

The establishment of monitoring relations within a network $Net = (N, M, s)$ can be based on different aspects. Therefore, the suitability function s has to be defined accordingly. Note that the suitability information typically is not computable before nodes receive information from other nodes. If it is for instance desired that nodes should be monitored by nodes with a similar hardware equipment and a fast network connection, the suitability function could be set to $s(u, v) = \frac{h(u,v)+n(u,v)}{2}$ where $h(u, v)$ returns a value within $[0, 1]$ indicating the similarity of the hardware equipment of u and v and $n(u, v)$ returns a value within $[0, 1]$ indicating the performance of the network connection. Such a scenario would make sense if a fast network connection improves the monitoring quality and in the case of an outage of a node, another node with similar hardware equipment is likely to have the ability to inherit the tasks of the failed node. Thus, the setting of the suitability function influences the establishment of monitoring relations. The definition of a suitability function should reflect the requirements of a mon-

itoring system. All relevant factors should be included and weighted according to its importance.

Now three algorithms to establish monitoring relations in an autonomous distributed way are presented: INDIVIDUAL, which constructs individual monitoring relations, MERGE and SPECIES which install monitoring groups.

The idea of INDIVIDUAL is very simple: each node tries to identify the m most suitable nodes and asks them to monitor it.

In the initial state of the algorithm MERGE, each node forms a group only consisting of itself as leader node. Groups merge successively until they reach a size greater than m.

SPECIES distinguishes between the two species *leader* and *non-leader*. The specificity of a node is random-driven. Non-leaders try to join a group and each group is controlled by one leader. In the case of an inadequate ratio of leaders to non-leaders, nodes can change its specificity.

Individual

Individual monitoring relations denote monitoring responsibilities set individually for each node. Using the suitability function, nodes can identify suitable surveillants. The most suitable ones are asked to monitor it. Therefore, nodes send monitoring requests to other nodes and wait for their acknowledgement. This process is repeated until the node has established m acknowledged monitoring relations. In Algorithm 10, the above described algorithm is formalised as pseudocode.

A further requirement for individual grouping algorithms which is omitted here could be that each node u monitoring a node v needs to know all other nodes also monitoring v, i.e. if $u \rightarrow v$ then u needs to know the set $\overset{*}{\rightarrow} v$. This might be necessary as in the case of a failure of v, all monitoring nodes could e.g. have to hold some kind of vote to gather a consistent view and to plan repairing actions respectively. This feature of closed monitoring groups could easily be integrated into INDIVIDUAL. This has not been done in order to investigate the more general algorithm as stated here.

Merge

In this section the MERGE algorithm is discussed which establishes closed monitoring groups. Within these groups all nodes monitor each other. Every group has a group leader. Typically, the initial situation is a monitoring network $Net = (N, S, \varnothing)$ without monitoring relations and a number m

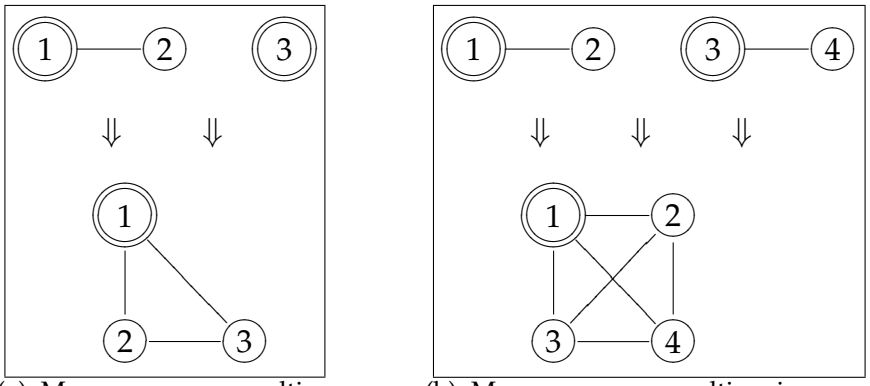
Algorithm 10 INDIVIDUAL

```
1:  id                                                          ▷ the id of this node
2:  m                                                           ▷ number of surveillants
3:  𝒩                                                           ▷ set of known nodes
4:  id ⟶*= ∅                                                    ▷ set of monitored nodes
5:  ⟶* id = ∅                                                   ▷ set of surveillants
6:
7:  loop
8:      if received message msg from n then
9:          if type of msg is 'request' then
10:             id ⟶* = id ⟶* ∪ {n}
11:             send('ack', id) to n
12:         end if
13:         if type of msg is 'ack' then
14:             ⟶* id = ⟶* id ∪ {n}
15:         end if
16:     else
17:         if |⟶* id| < m then
18:             select most suitable node n out of 𝒩 \ ⟶* id
19:             send('request', id) to n
20:         end if
21:     end if
22: end loop
```

which determines the desired number of surveillants. During the grouping of the nodes into monitoring groups, existing groups smaller than $m + 1$ merge with other groups until the resulting group has enough members. Due to this mechanism the maximal size can be limited by $2 \cdot (m + 1) - 1$. If a group of size $2 \cdot (m + 1)$ exists, it can be split into two groups of valid size $m + 1$. The group leaders which belong to a monitoring group smaller than $m + 1$, ask other suitable group leaders to merge their groups. If this request is accepted the groups merge and the requesting group leader must give off its leadership. The requested group leader is the leader of the newly formed group. After such a merging process the group leader informs all members about the new group. Nodes which lost the leadership adopt a completely passive role in the further grouping process and are not allowed to accept merging requests from other leaders anymore.

Let us consider an example where m is 2, i.e. groups of minimum size 3 are formed. In Figure 3.3(a) two groups are examined, one consisting of Nodes 1 and 2 in which Node 1 is leader and the other group consisting only of Node 3. Node 3 is requesting the group of Node 1 to merge. After the merge process a new group is formed with exactly 3 members and node 1

is leader of that group. Merge requests are never denied by leaders. Thus, as you can see in Figure 3.3(b), it is possible that groups emerge which have more than the desired $m + 1$ members.



(a) Merge process resulting in desired group size

(b) Merge process resulting in oversized group

Figure 3.3: MERGE *scenarios*

If groups become greater than or equal to $2 \cdot (m + 1)$, as illustrated in Figure 3.4, a splitting is performed resulting in two monitoring groups which both have at least $m + 1$ members, what is enough to stop active merging activities. Thus the resulting group sizes of the MERGE algorithm are always between $m + 1$ and $2 \cdot (m + 1) - 1$.

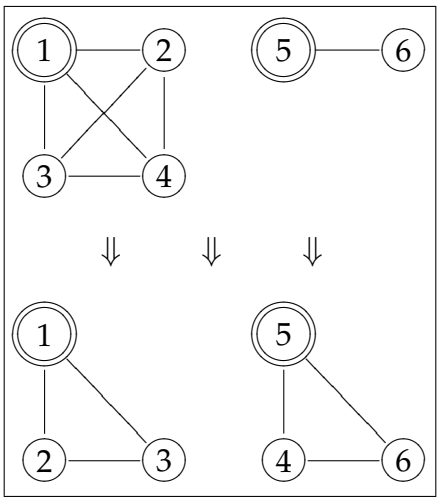


Figure 3.4: *Merge and consecutive split*

In Algorithm 11, the described grouping algorithm is formalised as pseudocode. Please note that only the most interesting parts of the algorithm are presented, due to space limitations. For instance, the notification of group members when the group has changed is omitted.

Algorithm 11 MERGE

```
1:  id                                                  ▷ the id of this node
2:  m                                                   ▷ minimum number of surveillants
3:  𝒩                                                   ▷ set of known nodes
4:  𝒢 = {id}                                            ▷ set of group members
5:  l = id                                              ▷ leader, initially set to node id
6:  wr = F                                              ▷ is the node is waiting for a response
7:
8:  loop
9:      if received message msg from n then
10:         if type of msg is 'request' then
11:             if id = l then
12:                 if wr then send ('waiting', id) to n
13:                 else
14:                     if |𝒢| + |msg.𝒢| ≥ 2 · (m + 1) then
15:                         H = choose ⌊(|𝒢|+|msg.𝒢|)/2⌋ − |msg.𝒢|
16:                               group members to handover
17:                         send ('handover', H) to n
18:                     else
19:                         𝒢 = 𝒢 ∪ msg.𝒢
20:                         send ('ack',𝒢) to all 𝒢 \ {id}
21:                     end if
22:                 end if
23:             else
24:                 send ('non-leader', 𝒢) to n
25:             end if
26:         else if type of msg is 'ack' then
27:             l = n
28:             𝒢 = msg.𝒢
29:             wr = F
30:         else if type of msg is 'handover' then
31:             𝒢 = 𝒢 ∪ msg.H
32:             wr = F
33:         else if type of msg is 'non-leader' then
34:             store information that n is no leader
35:             wr = F
36:         else if type of msg is 'waiting' then
37:             affects the selection of most suitable node
38:             wr = F
39:         end if
40:     else
41:         if id = l ∧ |𝒢| < m + 1 then
42:             select most suitable node n out of 𝒩 \ →* id
43:             send('req', 𝒢) to n
44:             wr = T
45:         end if
46:     end if
47: end loop
```

Species

Like the MERGE algorithm, SPECIES also installs closed monitoring groups. It is based on the existence of two species: *leader* and *non-leader*. Leaders are group managers and each group contains exactly one leader. Non-leaders contact the most suitable leader trying to join its group. The specificity of a node is random-driven and dependent on the value of m. Consider a network consisting of n nodes. The optimal number of leaders is $\frac{n}{m+1}$, as the following example illustrates: Within a small network of 12 nodes, closed monitoring groups need to be installed with $m = 2$, i.e. two surveillants per node or groups of size three. The optimal case for that are four groups of size three. Thus $\frac{n}{m+1} = \frac{12}{3} = 4$ leaders are needed which the non-leaders can join. Therefore, the SPECIES algorithm selects every node as leader with probability $\frac{1}{m+1}$. As it is worse to have too many leaders than too few, the probability of a node to become a leader can be adjusted to e.g. $\frac{0.8}{m+1}$. However, the random assignment of species to nodes does not guarantee a valid distribution into leaders and non-leaders. Thus, if a leader recognises that there are too many of them, they can toggle their species and transform into a non-leader. Vice versa, if nodes cannot find leaders to join they transform into a leader with a certain probability.

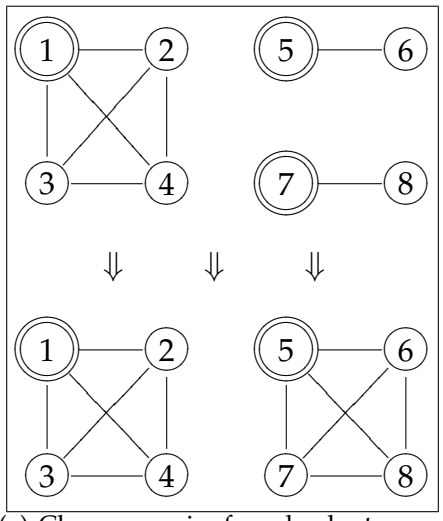
The network shown in Figure 3.5(a) contains too many leaders. In this case m is 3 which means groups of sizes of at least 4 need to be formed. However, this is not possible in this example. If no non-leader joins the groups smaller than 4, their leaders try to contact other leaders in order to find groups with enough members to poach some non-leaders. If this also fails, leaders then transform to non-leaders with a certain probability. This happens with Node 7 in this example. After that transformation a valid grouping is possible.

Figure 3.5(b) shows the contrary situation as above, where too few leaders are available, in this case even none. If non-leaders are unable to find any leader, they become leader with a certain probability.

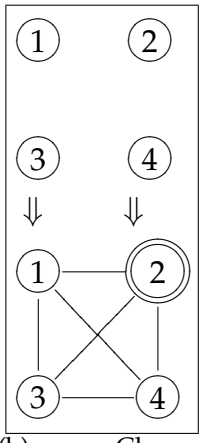
There are two cases how non-leaders join a group. If they do not belong to a group yet, they themselves care to find a group and join it. The other case is shown in Figure 3.6. Leaders controlling an undersized group try to find oversized groups and ask their leaders to handover unneeded members.

In Algorithm 12, the most important parts of SPECIES are formalised as pseudocode.

After the introduction of the proposed grouping algorithms, an evaluation is provided in the following section.



(a) Change species from leader to non-leader



(b) Change species from non-leader to leader

Figure 3.5: SPECIES *scenarios*

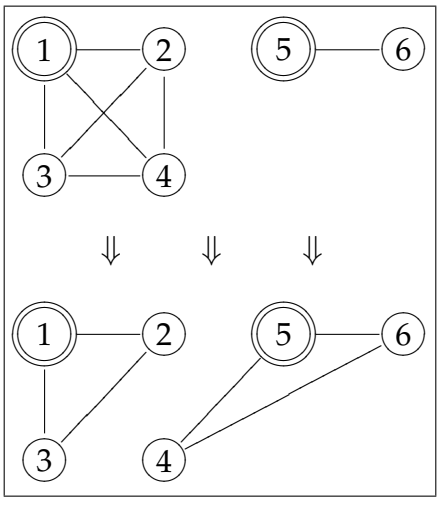


Figure 3.6: *Handover of group members*

Algorithm 12 SPECIES

id ▷ the id of this node
m ▷ minimum number of surveillants
$\mathcal{N}$ ▷ set of known nodes
$\mathcal{G} = \{id\}$ ▷ set of group members
$l = \begin{cases} id & with\ probability\ \frac{0.8}{m+1} \\ undefined & otherwise \end{cases}$
$wr = F$ ▷ is the node waiting for a response

loop
  if received message msg from n **then**
    if type of msg is 'request' **then**
      if $id = l$ **then**
        $\mathcal{G} = \mathcal{G} \cup \{n\}$
      end if
    else if type of msg is 'handover-request' **then**
      if $id = l$ **then**
        if wr **then**
          send ('waiting', id) to n
        else
          x = min($\lfloor \frac{|\mathcal{G}|+|msg.\mathcal{G}|}{2} \rfloor - |msg.\mathcal{G}|, |\mathcal{G}| - (m+1)$)
          H = choose x group members to handover
          send ('handover', H) to n
        end if
      else
        Forward message to random node, preferably a leader
      end if
    else if type of msg is 'handover' **then**
      $\mathcal{G} = \mathcal{G} \cup msg.H$
      $wr = F$
    else if type of msg is 'chgspecies' **then**
      if $id = l$ **then**
        $\mathcal{G} = msg.\mathcal{G}$
      else
        Forward message to random node, preferably a leader
      end if
    else if type of msg is 'waiting' **then**
      affects the selection of most suitable node
      $wr = F$

Algorithm 12 SPECIES - continued

```
38:         else
39:             if id ≠ l ∧ |𝒢| ≤ 1 then
40:                 select most suitable node n out of 𝒩
41:                 send('req', 𝒢) to n
42:                 l = n
43:             end if
44:             if id = l ∧ |𝒢| < m + 1 ∧ with probability of e.g. 25% then
45:                 if other suitable leader n is known then
46:                     send('handover-request', 𝒢) to n
47:                 else if no leader can handover nodes then
48:                     change species to non-leader
49:                     l = undefined
50:                     send('chgspecies', 𝒢) to random node, preferably a
51:                                                                 leader
52:                 end if
53:             end if
54:             if id = l∧ no leader known ∧ probability of e.g. 25% then
55:                 change species to leader
56:                 l = id
57:             end if
58:         end if
59:     end if
60: end loop
```

3.6 Evaluation

In this section an evaluation for the above introduced algorithms is provided. For the purpose of evaluating and testing, a toolkit has been implemented which is able to simulate distributed algorithms based on message passing. It is written in JAVA and allows the construction of networks consisting basically of nodes, channels which connect two nodes, and algorithms running on nodes. As the simulation runs on one single computer, a random strategy selects the next node whose algorithm is executed partially. Thus, the asynchronous behaviour of distributed systems is covered. It is assumed that the communication channels do not drop messages and deliver them in the correct order.

The nodes of the monitoring network $Net = (N, M, s)$ used for the evaluation are theoretically arranged as a grid, as shown in Figure 3.7 for an example network consisting of 100 nodes. The nodes of the network are labelled with natural numbers which represent their ID. Note that the algorithms are neither based on that fact nor take any advantage of that. The distance of two nodes u, v within the grid determines their mutual monitoring ability. Thus, the suitability has been set to the reciprocal value of the Euclidean distance of the nodes within the grid.

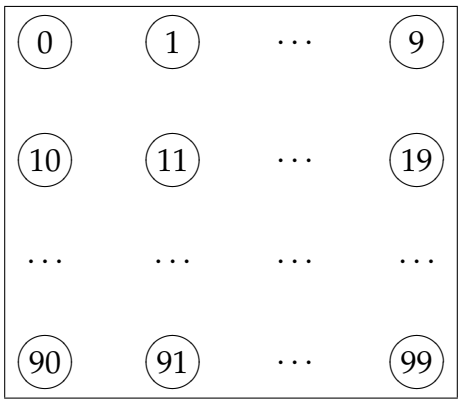


Figure 3.7: *Evaluation network of 100 nodes*

The evaluation network consists of 1000 nodes[1], where all nodes are able to communicate with each other. However, in most evaluation scenarios the nodes only have sufficient information about a certain number of nodes to compute a suitability value. This models the concept that in many networks nodes do not know everything but have a limited view.

The introduced grouping algorithms are evaluated within different scenarios. The evaluation focuses on the scalability of the establishment of moni-

[1]Except for the measurements of the scalability regarding the network size where the number of nodes have been varied.

toring relations, the optimality of the relations regarding the suitability metric, and the failure tolerance of a system if failure detectors are used together with the grouping approach. The evaluations have been conducted using different sets of parameters like the values for the desired number of surveillants m and the amount of information about other nodes. Each evaluation scenario has been replayed 1000 times and the results have been averaged.

Recall, a monitoring network Net is represented as (N, M, s), where N is the set of nodes of a network, $M \subseteq N \times N$ is the monitoring relation, and s is a function from $N \times N$ to a real value within $[0, 1]$. The task of a grouping algorithm is, given a positive integer $m < |N|$, to establish monitoring relations such that every node of the network is monitored by at least m nodes.

As suitability function $s(u, v)$, the reciprocal value of the Euclidean distance of the nodes u and v is used. At the beginning, the monitoring relation is empty, i.e. $M = \emptyset$. This means that the network is in a state where no monitoring relations are established yet. To model the fact that nodes usually do not have a complete view of the whole network, the value κ describes the part of the network each node is aware of. A value of $\kappa = 10$ means that each node has information about 10 randomly chosen nodes.

In the following the results of the conducted evaluations are presented.

3.6.1 Scalability

To establish monitoring relations, messages need to be sent. In the following, this overhead is evaluated for the proposed grouping algorithms. All experiments have been conducted according to the description given above.

First the scalability regarding the network size is evaluated. In this experiment the number of desired surveillants m is set to 5 while each node knows 50 other nodes, i.e. $\kappa = 50$. Figure 3.8 shows the results of this experiment. The values on the x-axis stand for the network size and the average number of messages sent by each node is depicted on the y-axis.

As m is 5, each node executing the INDIVIDUAL algorithm needs 10 messages, 5 monitoring requests and 5 responses. MERGE needs less than 6 messages, SPECIES less than 4. The results indicate that all three algorithms can be classified as being independent from the network size, as the nodes basically do not send more messages within a bigger network. The algorithm SPECIES performs even better in bigger networks. The reason for this behaviour is the random-driven determination of the specificity. The aim of

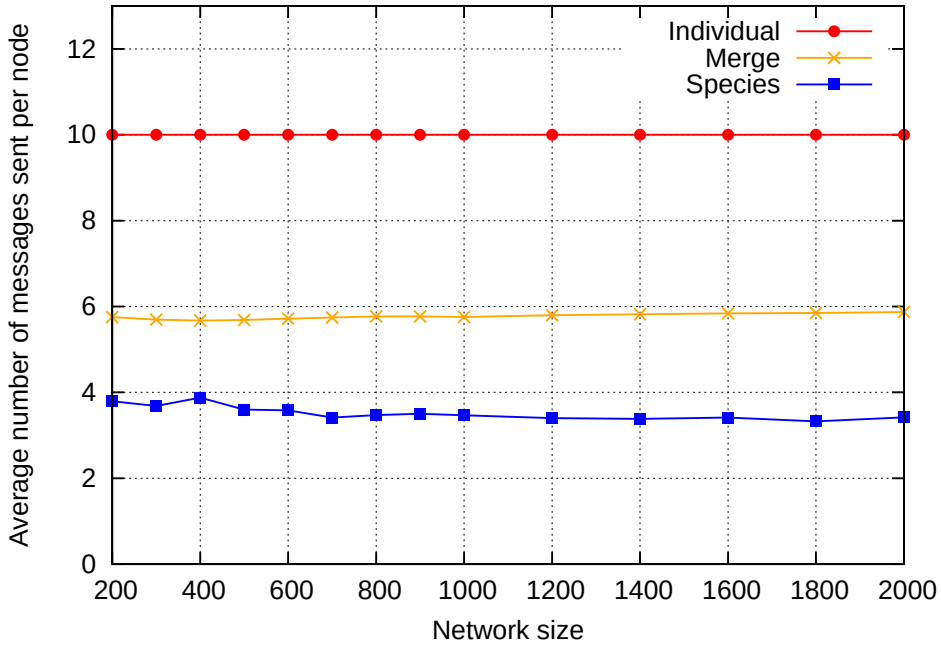


Figure 3.8: *Scalability of grouping algorithms regarding network size ($\kappa = 50$)*

that process is to achieve a division into leaders and non-leaders of a defined ratio. In general, the bigger the network the better this ratio is met.

Thanks to the independence of the overhead caused by the grouping algorithms from the network size, all introduced algorithms seem suitable to be applied within complex distributed systems.

All following evaluations have been conducted with a network size of 1000 nodes. To evaluate the overhead with regard to the sizes of the formed groups, the message sending behaviour of the algorithms is compared using different values for the minimum group size m within $[3, 4, \ldots, 20]$, and $\kappa = 100$.

Figures 3.9 shows the results of that experiment. It depicts the average number of sent messages on the y-axis. The x-axis stands for the different values of m. Figure 3.10 presents exactly the same information as Figure 3.9, but without the data of INDIVIDUAL to provide a better comparison possibility between MERGE and SPECIES.

The SPECIES algorithm manages to group with the least messages of the three algorithms. The number of sent messages is independent from the number of surveillants. INDIVIDUAL scales linearly with the number of surveillants. The number of messages for MERGE is strictly increasing, but only logarithmically due to the exponential group growth based on merging.

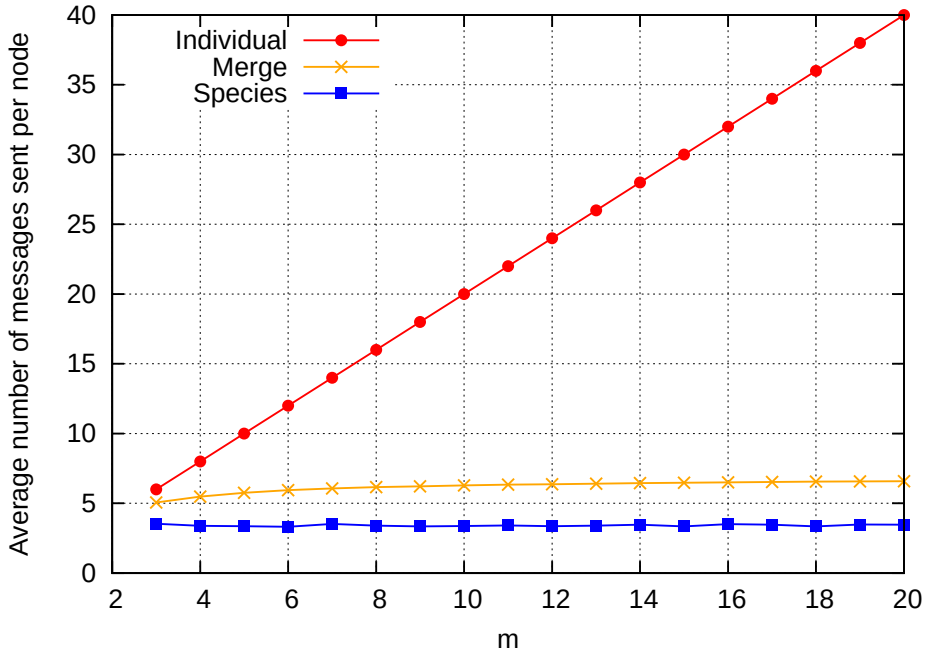


Figure 3.9: *Scalability of grouping algorithms regarding group size ($\kappa = 100$)*

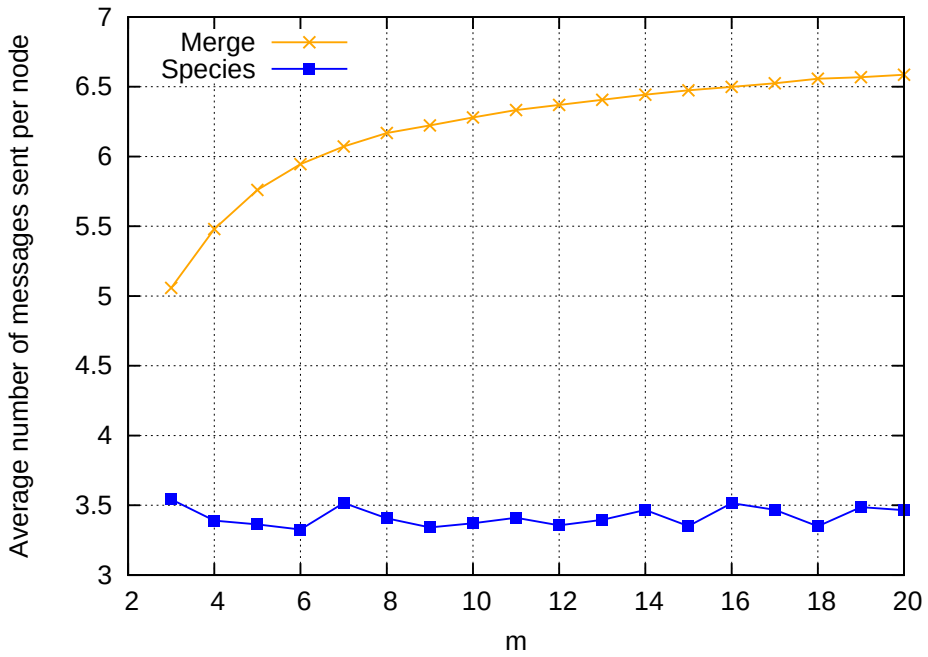


Figure 3.10: *Scalability of grouping algorithms regarding group size - without* INDIVIDUAL

The three algorithms differ in the way they are able to meet the desired number of surveillants m. As it is mandatory to install at least m monitoring relations per node, only greater or equal values for the resulting group

sizes are possible. Figure 3.11 shows the resulting number of surveillants in comparison to the value of m. INDIVIDUAL manages to install exactly m surveillants per node. For closed monitoring groups it is a much harder problem to exactly meet this condition. MERGE and SPECIES typically form slightly larger groups in order to allow for a fast and robust grouping process.

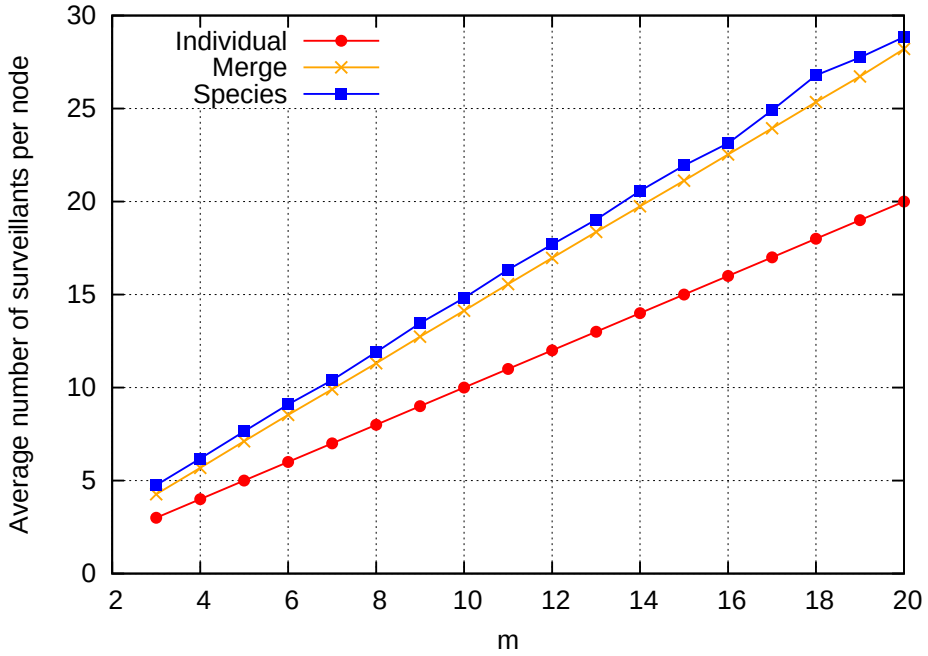


Figure 3.11: *Resulting group sizes caused by different values for m*

The next section examines the monitoring relations with respect to their suitability according to the suitability function.

3.6.2 Suitability

As stated in the specification, the algorithms are supposed to take the suitability of the nodes into account. This means the term

$$\sum_{v \in N} \sum_{u \in \overset{*}{\rightarrow} v} s(u, v) \tag{3.1}$$

should be maximised. The average suitability within the evaluation network is about 0.09. This means a random grouping produces monitoring relations of about that value. Figures 3.12 to 3.15 show the results of the experiments concerning the suitability of the algorithms for different values of κ (10, 50, 100, 1000). This parameter represents the size of the nodes'

view on the network. The x-axis represents the number of surveillants per node, the y-axis depicts the average suitability of the formed groups based on Formula 3.1.

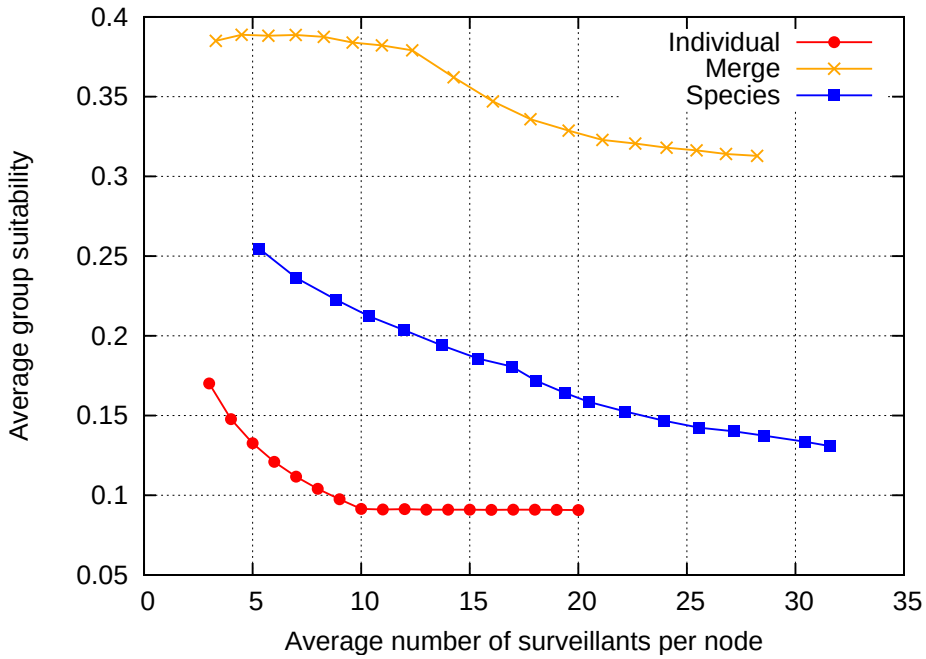


Figure 3.12: *Suitability of grouping algorithms* ($\kappa = 10$)

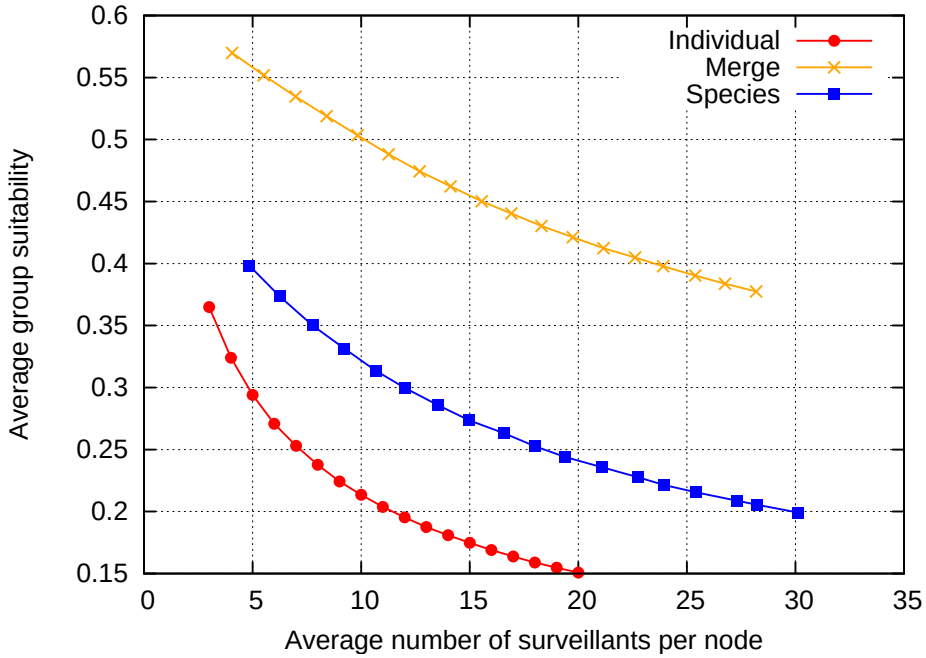


Figure 3.13: *Suitability of grouping algorithms* ($\kappa = 50$)

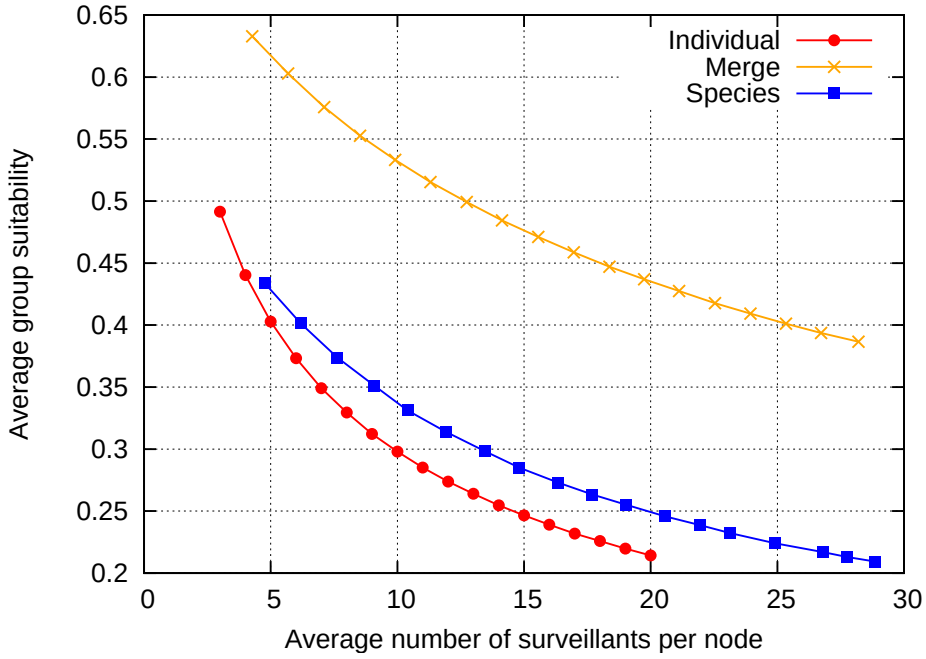


Figure 3.14: *Suitability of grouping algorithms ($\kappa = 100$)*

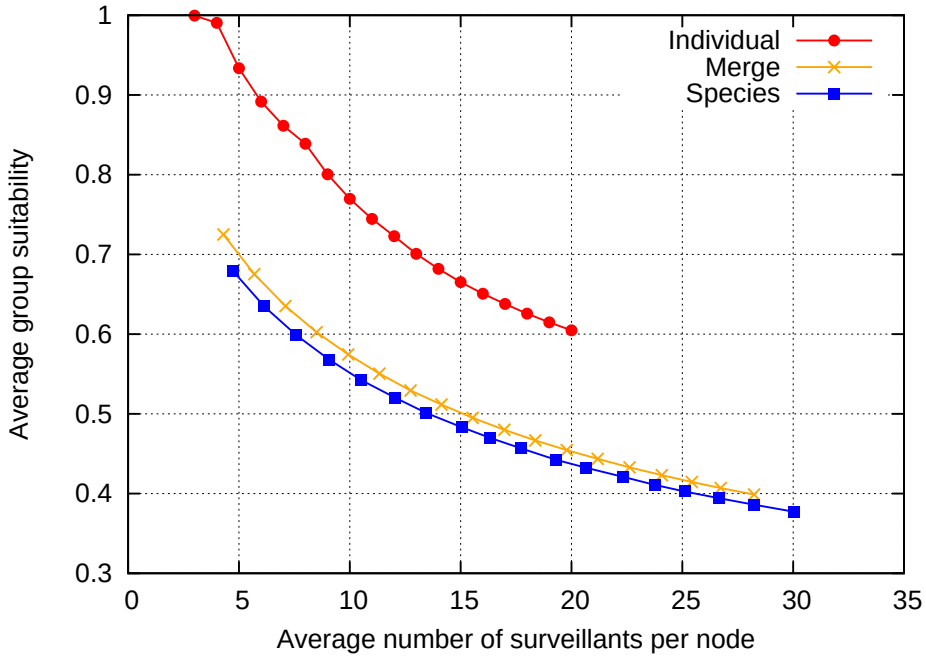


Figure 3.15: *Suitability of grouping algorithms ($\kappa = 1000$)*

For all algorithms holds the obvious fact that bigger group sizes cause lower values for the suitability. It is remarkable that SPECIES and especially MERGE handle grouping with limited information very well. In the case of

full information about the network ($\kappa = 1000$) INDIVIDUAL performs optimal, as each node is able to identify and request its most suitable surveillants.

Figure 3.16 illustrates all results of the suitability measurements at once. It shows on the x-axis the number of surveillants, on the y-axis the number of known nodes κ, and on the z-axis the resulting suitability values. The image on the top left shows the results for INDIVIDUAL, on the top right for MERGE, on the lower left for SPECIES, and on the lower right a comparison.

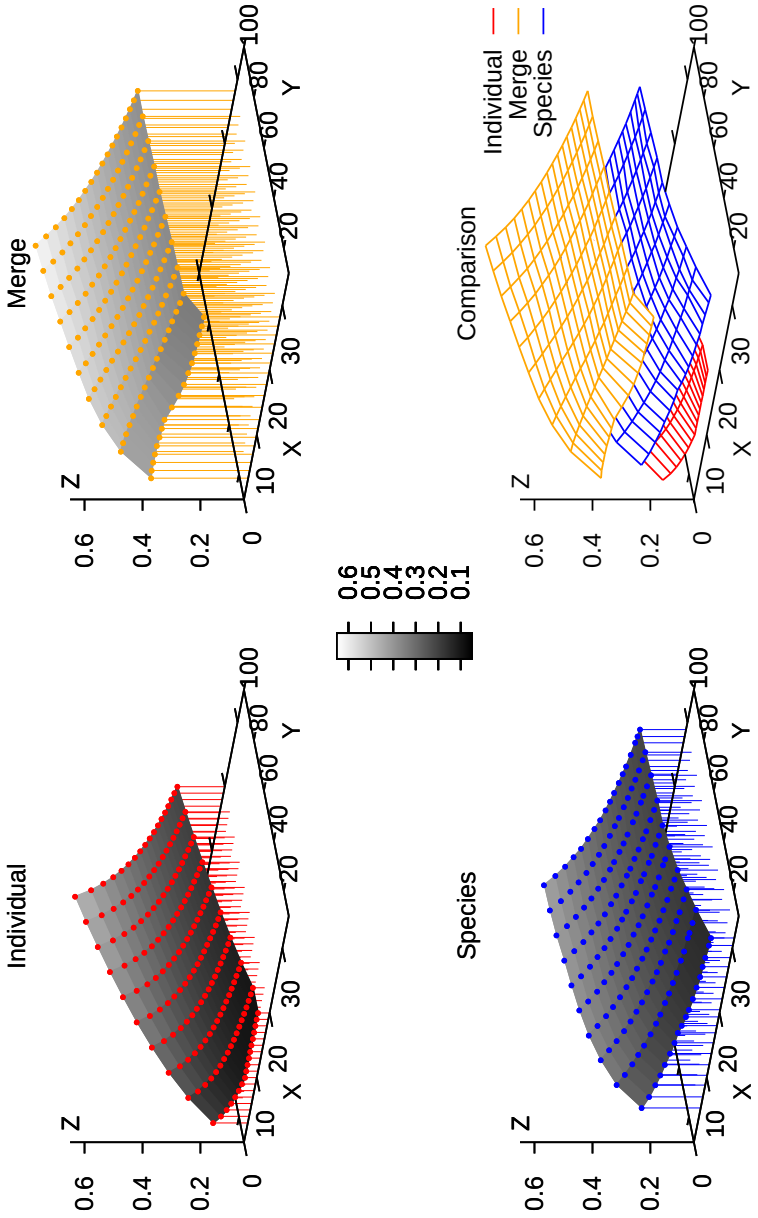


Figure 3.16: *Suitability of grouping algorithms*

3.6.3 Failure tolerance

In this section, the gain of applying the proposed grouping techniques with respect to failure tolerance is investigated. To evaluate the failure tolerance of the monitoring relations, the following methodology is used: It is assumed that a certain percentage of randomly chosen nodes within the network fail simultaneously, i.e. they crash and do not recover. Using failure detectors, nodes monitor each other according to the installed monitoring relations by a grouping algorithm. It is assumed that failure detectors eventually detect the failure of a node. An undetected failure means the unrecognised failure of a node. In this setting this is only possible if a node and all its surveillants fail simultaneously.

The detection of a failure is the prerequisite of a subsequent repair or self-healing respectively. If a node has no surveillant, its failure equals an undetected failure. If in a network any node monitors all other nodes, only the complete failure of the whole network results in undetected failures. However, for more complex systems, the latter monitoring strategy typically introduces an excessive overhead.

Before the evaluation results are presented, a short view on failure tolerance motivated by probability theory is given.

Let X be the number of elements within a set, $Y \leq X$ the number of elements within this set possessing a feature F, and $x \leq X$ the number of elements which are randomly chosen from the set. The probability of k elements with feature F to be in the randomly chosen set is then

$$\frac{\binom{Y}{k}\binom{X-Y}{x-k}}{\binom{X}{x}},$$

according to the hypergeometric distribution [Fel70].

Considering a network $Net = (N, M, s)$ and a number of surveillants per node of $m < |N|$. If ϕ random nodes of the network fail, where $m+1 \leq \phi \leq |N|$, the probability for the undetected failure of a certain node is

$$\frac{\binom{m+1}{m+1}\binom{|N|-m+1}{\phi-m+1}}{\binom{|N|}{\phi}} = \frac{\binom{|N|-m+1}{\phi-m+1}}{\binom{|N|}{\phi}}.$$

If ϕ is lower than $m+1$, the probability for an undetected failure is obvi-

ously 0. If for instance $\phi = 10\%$ of the nodes of a network $Net = (N, M, s)$ consisting of 100 nodes fail, and each node is monitored by $m = 3$ nodes, then the probability for a certain node $\eta \in N$ to fail undetectably is

$$\frac{\binom{|N| - m + 1}{\phi - m + 1}}{\binom{|N|}{\phi}} = \frac{\binom{100 - 4}{10 - 4}}{\binom{100}{10}} \approx 5 \cdot 10^{-5}.$$

The following simulations have been conducted as before with a network $Net = (N, M, s)$ of 1000 nodes. Monitoring relations are established with all three proposed grouping algorithms and different values for m. It is measured how many undetected failures occur if a certain percentage ϕ of random nodes fail.

Figure 3.17 presents the results for $\phi = 10\%$, Figure 3.18 for $\phi = 50\%$, and Figure 3.19 for $\phi = 90\%$. The x-axis shows the average number of surveillants per node, the y-axis the number of undetected failures.

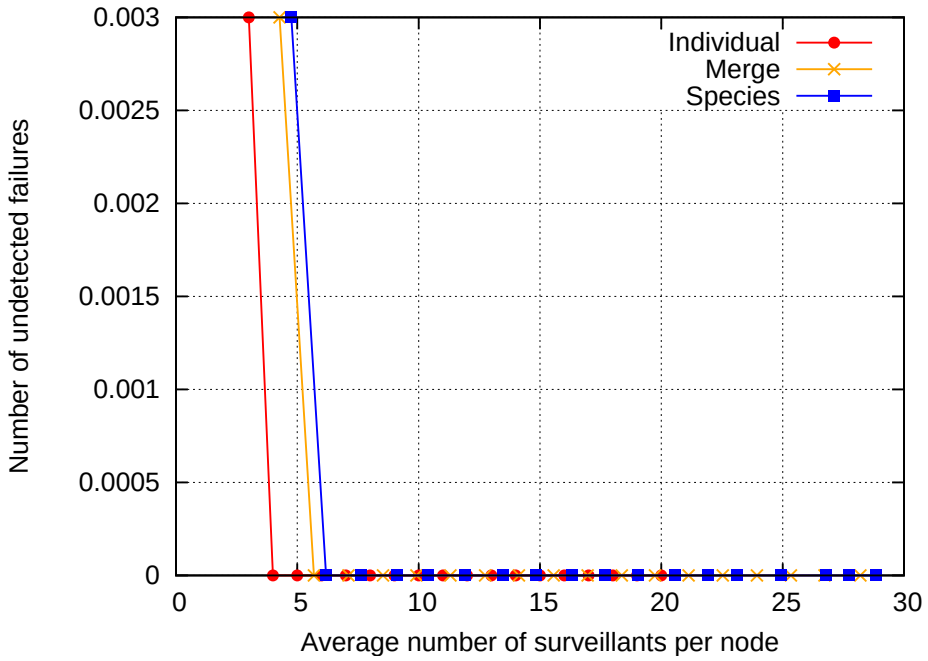


Figure 3.17: *Failure tolerance of grouping algorithms (10% failure)*

In all cases the number of undetected node failures decreases with a higher number of surveillants. It can be seen that, with a number of surveillants of 15, a failure of every second node in the network does not result in any undetected node failures. These results can be used as a utility to choose an adequate value for m, which is a balancing act between overhead and failure tolerance.

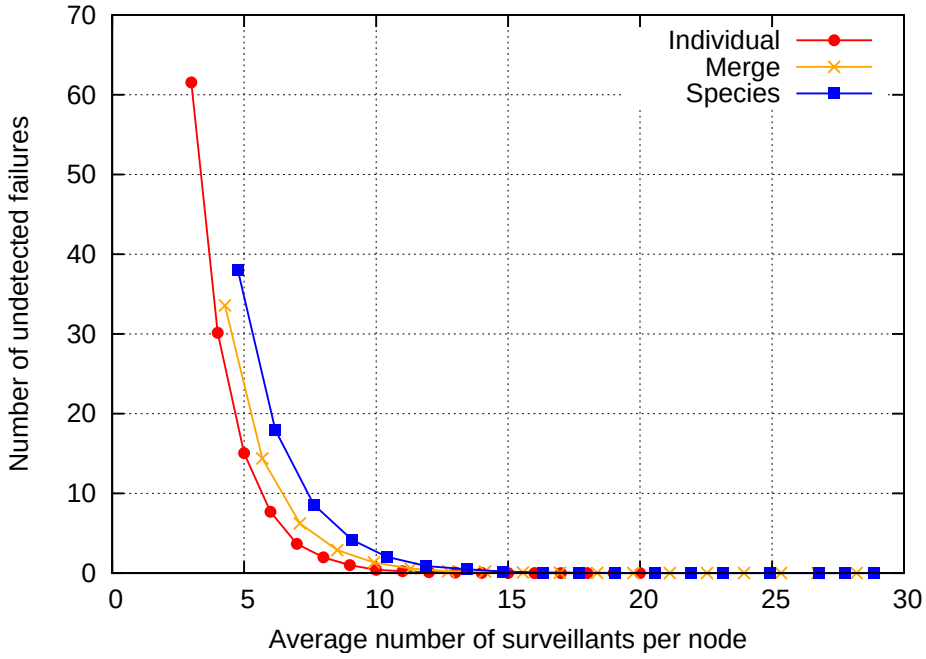


Figure 3.18: *Failure tolerance of grouping algorithms (50% failure)*

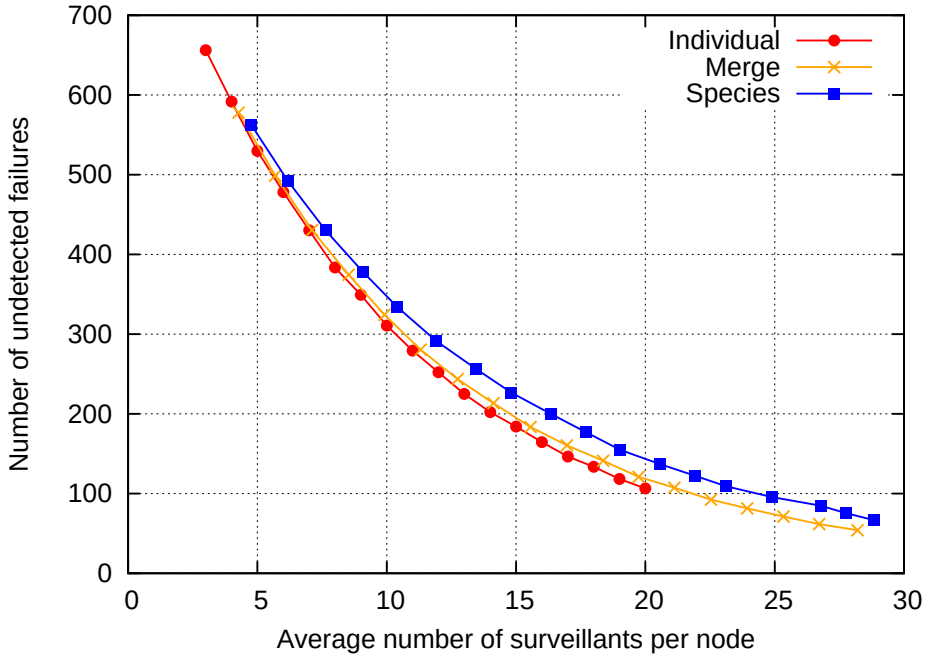


Figure 3.19: *Failure tolerance of grouping algorithms (90% failure)*

The algorithm INDIVIDUAL performs best in all three scenarios. The reason is that it has no variance in the number of surveillants. The parameter m exactly determines the resulting group size, i.e. the number of surveillants.

For closed monitoring groups this number varies. The value for m only determines the minimum number of surveillants. Thus, an average number of surveillants of e.g. 10 does not exclude groups of lower sizes. Consider for example a network of 10 nodes, arranged in two monitoring groups. In the first case two groups of sizes 5 (no variance), in the second case one group of size 4 and one group of size 6 (variance of 2). If for example only 4 nodes fail, in the latter case undetected failures are possible, in the former not. The variance of the group sizes impacts the number of undetected failures, while a low variance is better. INDIVIDUAL has no variance, SPECIES has the highest variance, and MERGE lies in between. The evaluation results reflect this fact.

While the influence of the variation of the number of surveillants is given, all three algorithms show the same basic behaviour, which is depicted in Figure 3.20 with the algorithm INDIVIDUAL. The x-axis shows the number of surveillants, the y-axis the percentage of node failures ϕ, and the z-axis the number of undetected failures. A high number of surveillants and a low percentage of node failures result in none or few undetected node failures. A low number of surveillants and many failed nodes result in a high rate of undetected failures.

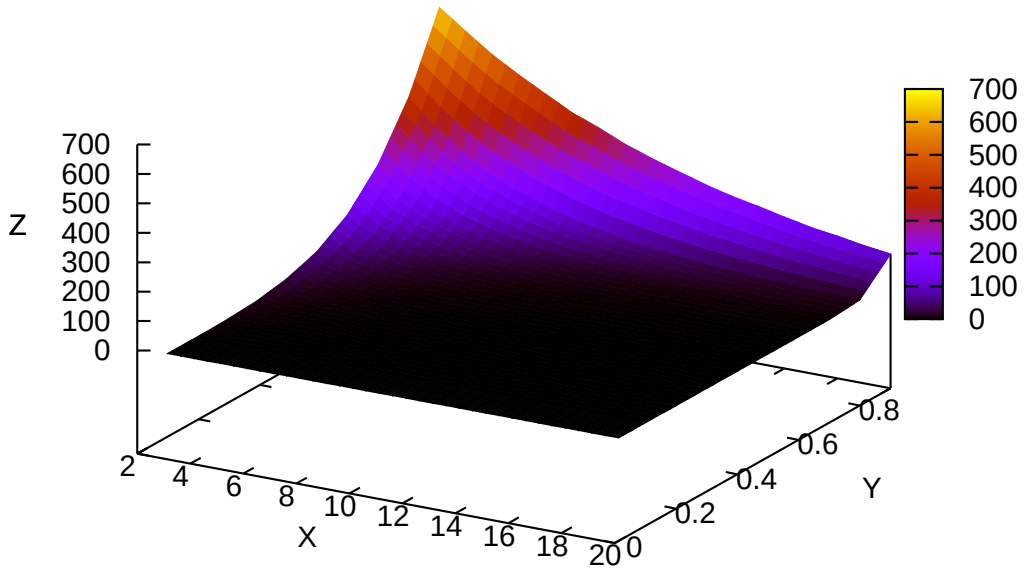


Figure 3.20: *Failure tolerance of* INDIVIDUAL

3.7 Conclusions and future work

In this chapter, the requirements for self-monitoring, distributed systems are presented. The task is to autonomously install monitoring relations, whereas the two types of individual monitoring relations and monitoring groups have been identified. This problem is novel and has been defined in a concise way. Three algorithms to install such monitoring relations are introduced and compared regarding their efficiency, suitability, and the failure tolerance they are providing. The algorithms are tailored to install monitoring relations very quickly, and this is important for self-healing systems. The evaluation shows that the overhead of all proposed algorithms is independent from the network size. Therefore, they are suitable for complex, large scale, distributed systems. Each algorithm needs only a very limited number of messages per node in order to fully install monitoring relations. Existing algorithms dealing with similar problems are either too complex for self-healing systems or unable to consider the suitability of monitoring relations. The conducted evaluations and theoretical considerations regarding the failure tolerance can be used to determine the necessary number of monitoring nodes. The formation of hierarchical groups could further improve the monitoring relations and could be a starting point for future work.

4
Failure recovery

4.1 Introduction

The self-healing concepts dealt with so far correspond to gather information about the state of the system. In this chapter a failure recovery engine is presented which autonomically controls the entities of a distributed system in order to maintain user-defined properties and to recover from unwanted conditions respectively. An automated planning approach is the basis of the proposed recovery engine.

Srivastava et al. [Sri05] argue that planning is an evolutionary next step for AC systems that use procedural policies. Policies are often used to specify the desired behaviour of computer systems. To meet these policies under all system conditions, lists of necessary actions can be defined. During runtime, a policy engine will verify the conditions and take the stipulated action. However, for self-healing in distributed systems the enumeration of all conditions, i.e. all possible types of failures is often impracticable. Also, the actions to recover the system can be too complex to specify manually. Furthermore, many different ways to recover a system from one certain fail state may exist, and this raises the complexity even more. A methodology that is better in line with the visions of OC and AC would be policies that only describe desired conditions that a system can meet by itself. Automated planning is a concept to enable computer systems to reason about actions in order to automatically compute plans to follow the policies.

As an overview of related work, failure recovery techniques are presented

in the next section. Section 4.3 gives an introduction to automated planning. Then, in Section 4.4, the proposed failure recovery engine is presented while Section 4.5 provides an evaluation of that approach. Finally, Section 4.6 concludes this chapter and outlines future work.

4.2 Related work

Failure recovery in distributed systems is a multifaceted area of computer science. In this section, a few interesting workings related with this topic are presented. The main focus lies on presenting work that has a connection to the approach proposed later on in this chapter. Commonly used techniques like replication, checkpointing, or message logging are assumed to be known and not introduced separately. For the interested reader it is referred to e.g. [CDK00] for some fundamental concepts on failure recovery in distributed systems.

Porter et al. [PTC06] propose an approach to generically achieve repair in overlay networks. They separate the generic and specific aspects of overlay repair to support and harness diverse environments. The repair is done in a localised manner as only nodes in the locality of a failed node are involved in the repair. This serves to guarantee the scalability of the approach. The authors do not treat individual nodes as the unit of failures, but more generally failed sections. Porter et al. propose a three-phase repair algorithm: In the first phase each node that detects the failure of a neighbouring node constructs its own view of the failed section. In phase two the nodes agree on a common view of the failed section and select a coordinator for the repair. In the third phase the coordinator performs the repair under monitoring of the remaining nodes.

Joshi et al. [JSHS05] deal with automatic model-driven recovery in distributed systems. When a failure of a component is detected, a recovery controller is invoked. The controller uses Bayesian estimation to combine monitor outputs and determine their likelihood. It then invokes one of two recovery algorithms to choose appropriate recovery actions: SSLRecover and MSLRecover. SSLRecover (Single Step Lookahead) is a computationally efficient greedy procedure that chooses the minimum cost recovery action that can restore the most likely fault hypothesis. MSLRecover (Multi-Step Lookahead) is a more sophisticated algorithm that looks multiple steps into the future.

Recovery Oriented Computing (ROC) [Fox02, Pat02] is a joint work of Stanford University and University of California, Berkeley. ROC takes the perspective that hardware faults, software bugs, and operator errors are facts to be coped with, not problems to be solved. It is proposed to reduce their

harmful effects by fast and graceful failure recovery. An indicator for system availability is $A = MTTF/(MTTF + MTTR)$, where MTTF is the mean time to system failure and MTTR the mean time to recovery after a failure. The target is to have a system with $A = 1$. The authors argue that in the past much effort has focused on achieving this by enlarging MTTF making hardware and software more reliable. The way ROC envisions to reach $A = 1$ is to focus on making MTTR much smaller than MTTF. In [Fox02] several reasons why to focus on recovery are cited, e.g. that human error is inevitable.

Arshad et al. [AHW04] use planning for failure recovery. Their approach automates failure recovery by capturing the state after failure, defining an acceptable recovered state as a goal and applying planning to get from the initial state to the goal state.

In this chapter a failure recovery based on automated planning is proposed. Before the technical realisation is explained, the following section gives an introduction to automated planning. If the reader is already familiar with that topic the next section may be skipped.

4.3 Introduction to Automated Planning

Automated planning is a branch of artificial intelligence. Its primary problem is the computation of sequences of actions which will achieve a specified goal from specified initial conditions. Domain-independent planning is concerned with the fundamental principles of planning as an activity. Domain-dependent planning deals with the application of planning within particular domains.

The conceptual model of a planning environment can be represented as *state transition system*. Formally, this is a four-tuple $\Sigma = (S, A, E, \gamma)$, where

- $\mathcal{S} = s_1, \ldots, s_{n^S}$ is a set of states,
- $\mathcal{A} = a_1, \ldots, a_{n^A}$ is a set of actions,
- $\mathcal{E} = e_1, \ldots, e_{n^E}$ is a set of events, and
- $\gamma : \mathcal{S} \times (\mathcal{A} \cup \mathcal{E}) \rightarrow 2^{\mathcal{S}}$ is a state transition function.

Basically, an automated planning problem can be described as follows:

Given a description of γ and the initial state $s_0 \in S$, find a sequence of actions that transforms the system from the initial state to a state s_g that satisfies the goal, i.e. $s_g \in S_g$ where $S_g \subseteq S$ is a set of goal states. The initial state is a snapshot of the system, any sequence of action that transforms the system to a goal state is called *valid plan*. Goals can also be called

objectives.

Planning algorithms often are based on some simplifying assumptions:

- The set of states $\mathcal{S}$ is finite.
- The system is fully observable, i.e. its state can be determined.
- The system is deterministic, i.e. every action or event has only one possible outcome.
- ...

The following section gives a short overview of different formalisms to represent planning problems.

4.3.1 Formal representation

A formalisation provides a formal description of all relevant information of planning problems. The system's state, the objectives, the actions, and its causalities have to be covered by such a formalisation.

Situation calculus [McC68, McC86] is a formalism to describe dynamic domains in first-order-logic. It is one of the most famous logic representations for planning problems. The system it depends on is modelled as consisting of a sequence of situations where each situation is a snapshot of the state of the system. Situations are a consequence from actions applied to previous situations. A binary function $Result(action, situation)$ returns the successor state of a state if a certain action is performed. Every predicate that can change from situation to situation obtains an extra situation argument. Actions are defined as axioms. These axioms are called *effect axioms*. The situation calculus needs some further axioms, called *frame axioms*. These determine which statements are not affected by the execution of an action.

Another approach for encoding a knowledge base for planners into first order predicate logics is the event calculus [KS86, MS99]. Situation calculus has two problems limiting its applicability [RN95]. It is not suitable to represent gradual change over time and spontaneous change. Event calculus can cope with these aspects. In the event calculus, events initiate periods during which certain properties hold. A property is initiated by an event and continues to hold until some event occurs that terminates it.

Expressing planning problems in logic is very expressive causing planning in such domains to be slow. The STRIPS (STanford Research Institute Problem Solver) [FN71] representation has been devised by Fikes and Nilsson to overcome with these computational difficulties. STRIPS is probably the most famous planning language. It was designed to control a robot named

Shakey. The STRIPS language describes states by a conjunction of function-free ground literals. Actions are modelled by operators. An operator consists of a precondition and an effect. The effects are often composed of an add list and a delete list. The semantic of an operator description states that an operation is only applicable if the precondition holds in the current system state and that after execution the literals of the add list will be added to the system state and the atoms of the delete list will be deleted.

In the following it is demonstrated how a planning problem can be modelled using a STRIPS like representation. Therefore the blocks world [ST96] is used. It consists of a set of blocks on a table. The blocks can be stacked and all blocks have the same size. Thus it is e.g. impossible to put two blocks directly onto another. There is one robot arm in the blocks world that has the ability to pick a block and put it either on the table or on another block. This can be modelled as follows:

The blocks on the table are constants. The four blocks of this example are described by the constants: A, B, C, and D. Furthermore, there are the predicates:

On(x,y) Indicates that block x is on block y.

Ontable(x) Is block x on the table?

Clear(x) True if no block is on top of x.

Armempty Is the robot arm empty or is it holding a block?

Holding(x) Indicates that block x is held by the robot arm.

The possible actions in the blocks world are modelled by four operators:

PICKUP(x) Takes a block from the table. The preconditions are that the block x is on the table, no other block is on top of x, and the robot arm is empty. The effects are that the arm is holding the block, the block is neither on the table nor clear, and the robot arm is not empty.

PUTDOWN(x) Puts a block on the table. The precondition is that the robot arm is holding a block. The effects are that the block x is on the table, clear, and the robot arm is empty.

STACK(x,y) Stacks block x on block y. The preconditions are that the robot arm is holding block x and block y is clear. The effects are that the robot arm is empty, x is on y, x is clear, the arm does not hold x, and y is not clear.

UNSTACK(x,y) Takes a the block x from block y. The preconditions are that the robot arm is empty, x is clear and is actually on y. The effects are that the arm is holding x, block y is clear, the arm is not empty, x is not clear and x is not on y.

```
PICKUP(x)
   precond:  Ontable(x), Clear(x), Armempty
   effect:   ADD: Holding(x)
             DEL: Ontable(x), Clear(x), Armempty

PUTDOWN(x)
   precond:  Holding(x)
   effect:   ADD: Ontable(x), Clear(x), Armempty
             DEL: Holding(x)

STACK(x,y)
   precond:  Holding(x), Clear(y)
   effect:   ADD: Armempty, On(x,y), Clear(x
             DEL: Holding(x), Clear(y)

UNSTACK(x,y)
   precond:  Armempty, Clear(x), On(x,y)
   effect:   ADD: Holding(x), Clear(y)
             DEL: Armempty, Clear(x), On(x,y)
```

Figure 4.1: *Operators of the blocks world*

Figure 4.1 shows the modelling of the operators in a STRIPS-like notation.

With the definition of the operations the properties of the domain are modelled. Given this set of operators, an initial state, and a goal, a STRIPS planner is able to generate a plan. Consider for example an initial state as in Figure 4.4(a) and a goal state as in Figure 4.4(b).

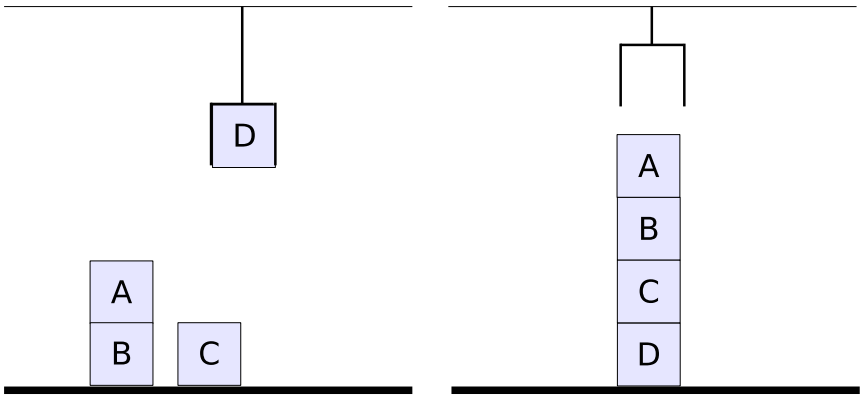
```
INIT
  Ontable(B) ∧ On(A,B) ∧ Clear(A) ∧
  Ontable(C) ∧ Clear(C) ∧ Holding(D)

GOAL
  On(C,D) ∧ On(B,C) ∧ On(A,B)
```

Figure 4.2: *Initial state and objective*

The accordant representation of this case in the terms of the predicates as introduced are shown in Figure 4.2.

Properties of the STRIPS language have been discussed over many years and a number of proposals for extensions and changes have been made, e.g. [Lif86, Rei01]. The ADL planning language [Ped89] is a more expressive language and solves some problems of the STRIPS formalism. Instead of the *Closed World Assumption* of STRIPS, i.e. negative information is not



(a) Initial state (b) Goal state

Figure 4.3: *Blocks world*

given explicitly, ADL is based on an *Open World Assumption*, i.e. undefined literals are unknown. Thus, state descriptions consist of positive and negative literals. Some further features of ADL are universal quantification of preconditions and effects, typing support, and the definition of context sensitive effects.

Like ADL, a number of planning languages emerged to extend the expressiveness of STRIPS in certain issues. As this wide range of formats makes it hard to compare the performance of planning algorithms, the PDDL 1.7 language [McD98] has been developed for the AIPS '98, The Fourth International Conference on Artificial Intelligence Planning and Scheduling Systems. PDDL is inspired by the STRIPS formulations, and basically a first-order logic language. The syntax is inspired by LISP, so much of the structure of a domain description is a LISP-like list of parenthesised expressions. The first version of PDDL included features of ADL [Ped89], SIPE-2 [Wil88], PRODIGY [Car88], and UCPOP [PW92].

The PDDL version for the AIPS in the year 2002 is called PDDL2.1 [FL03]. It contains new features, mainly connected with adding time to the language. Certain other features of the original language have been removed. The 2004 version is called PDDL2.2 [EH04]. It adds derived predicates and timed initial literals. The former are just backward-chaining axioms that allow a planner to achieve a goal by making the antecedent of one of them true. The latter are literals that will become true at a predictable time independent of what the planner does. PDDL3.0 [GL05] is the language for the 2006 competition and incorporates constraints and preferences.

PDDL separates the descriptions of actions that characterise domain behaviours from the description of specific objects, initial conditions, and goals that characterise a problem instance. A planning problem in PDDL consists of a domain description with a problem description. Figure 4.4

shows the domain description of the blocks world in PDDL.

```
(define (domain blocksworld)

  (:requirements :strips)
  (:predicates (clear ?x)
               (ontable ?x)
               (armempty)
               (holding ?x)
               (on ?x ?y))

  (:action pickup
    :parameters   (?x)
    :precondition (and (clear ?x) (ontable ?x) (armempty))
    :effect       (and (holding ?x) (not (clear ?x))
                       (not (ontable ?x)) (not (armempty))))

  (:action putdown
    :parameters   (?x)
    :precondition (holding ?x)
    :effect       (and (clear ?x) (armempty) (ontable ?x)
                       (not (holding ?x))))

  (:action stack
    :parameters   (?x ?y)
    :precondition (and (clear ?y) (holding ?x))
    :effect       (and (armempty) (clear ?x) (on ?x ?y)
                       (not (clear ?y)) (not (holding ?x))))

  (:action unstack
    :parameters   (?x ?y)
    :precondition (and (on ?x ?y) (clear ?x) (armempty))
    :effect       (and (holding ?x) (clear ?y) (not (on ?x ?y))
                       (not (clear ?x)) (not (armempty)))))
```

Figure 4.4: *Domain definition of the blocks world in* PDDL

A number of other planning representations exist, but as it goes beyond the scope of this work they are not further discussed here. The next section focuses on how to solve planning problems.

4.3.2 Planning techniques

The task of a planning technique is to find valid plans for given planning problems. Basically, planning can be seen as a search problem [NS72, AHT90]. Planning algorithms vary in the space that is searched, how the search is performed, in which way the plans are constructed, and so on. In the following an overview of different planning techniques is given.

State-space planning

The simplest classical planners are based on state-space planning where the search space is a subset of the state space. A state-space planning problem can be represented as a graph: Every node is an element of the set S, the states of the system. The edges are elements of A, the set of actions and are directed. The edges connect a state and the state that results from the application of the corresponding action at this state. A plan can be represented as a path within the graph. A path from the initial state s_0 to a goal state $s_G \in S_G$ is a solution of the planning problem. It is possible that multiple paths lead from the initial to a goal state. Figure 4.5 shows the search graph of the blocks world example.

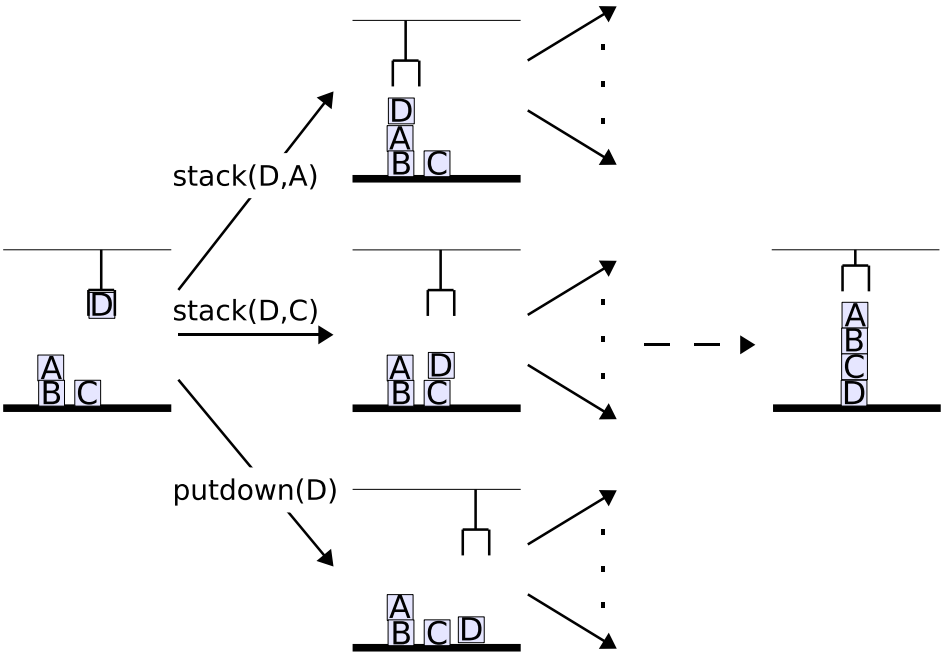


Figure 4.5: *State-space planning*

An advantage of representing the planning problem as a search in a graph is that general search algorithms can be applied [Kor88]. Dependent on

whether the planning algorithm is searching forwards starting from the initial state to the goal state or searching backwards from the goal state to the initial state it is called *progression planner* or *regression planner*. For typical planning problems regression search is faster than progression search, but unfortunately also more complicated [RN95]. Progression and regression search both are sound and complete, but state-space search has a rather high complexity as it produces a vast number of branches. State based planners normally output solutions as *totally ordered* sequences of actions, i.e. a list of actions.

The original STRIPS algorithm is a state-space regression planner with a mechanism to reduce the plan space significantly. However, this makes STRIPS incomplete, i.e. it is not guaranteed to find a solution for a problem even if one exists.

The FAST FORWARD (FF) planning system [Hof01] was the most successful automatic planner in the AIPS 2000. FF is based on forward search through the search space, guided by a sophisticated heuristic function to focus the search.

PRODIGY [Car88] is a system that incorporates forward state-space planning and learning capabilities. It uses control rules to guide the search - these rules may be general or domain specific, hand coded or automatically acquired, and may consist of heuristic preferences or definitive selections. If no control rules are defined, PRODIGY defaults to depth-first means-ends analysis. The learning process of PRODIGY refers to refining the possibly incomplete domain description through experience and learning knowledge to control the search process.

VVPLAN [VR99] is a forward state-space planner for the ADL language. It improves the performance by introducing a loop test relating to previously visited states.

Plan-space planning

An alternative to the search through the state-space is to search through the space of plans. In this space the search nodes represent partially specified plans, the start node is an empty plan which has no actions. The edges correspond to refinements of the partial plans, called *plan refinements*, that expand them until a complete plan has been created that solves the stated planning problem. Figure 4.6 illustrates a search through the plan space of the blocks world example.

To introduce some basic concepts of plan space planning a simple, banal planning problem is considered: preparing a lazy evening. The goal is to have beer, chips, and a movie, i.e. `HaveBeer` $\wedge$ `HaveChips` $\wedge$ `HaveMovie`, the

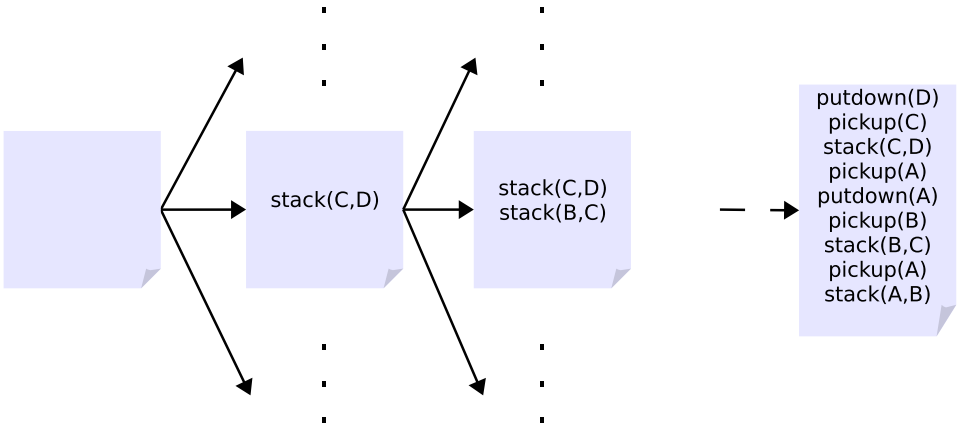


Figure 4.6: *Plan-space planning*

initial state is ¬HaveMoney ∧ ¬HaveBeer ∧ ¬HaveChips ∧ ¬HaveMovie. The operations are:

```
DRAWMONEY - effect: HaveMoney
BUYBEER   - precond: HaveMoney, effect: HaveBeer
BUYCHIPS  - precond: HaveMoney, effect: HaveChips
RENTMOVIE - precond: HaveMoney, effect: HaveMovie
```

Plan-space refinement algorithms often apply a principle called *least commitment* what means to defer decisions as long as possible. Instead of committing to a totally ordered sequence of operations as typical e.g. in state-space planning, plans are represented as partially ordered sequences where ordering restrictions are inserted only if necessary. Planners using partially ordered sequences of actions are called *partial order planners*, in contrast to *total order planners*. A total order plan is a *linearisation* of a partial order plan if it does not violate any of its ordering constraints. Figure 4.7 illustrates a partial order plan for the lazy evening planning problem, Figure 4.8 shows the linearisations of this partial order plan.

Partial order plans can be represented as a quadruple $\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL}, \mathcal{B} \rangle$ where

- $\mathcal{ST}$ is a set of steps,
- $\mathcal{O}$ is a set of ordering constraints,
- $\mathcal{CL}$ is a set of causal links, and
- $\mathcal{B}$ is a set of variable binding constraints.

$\mathcal{ST}$ contains the steps of the plan. These steps consist of the available actions $\mathcal{A}$ of the problem description. $\mathcal{O}$ is a binary relation over $\mathcal{ST}$. Its

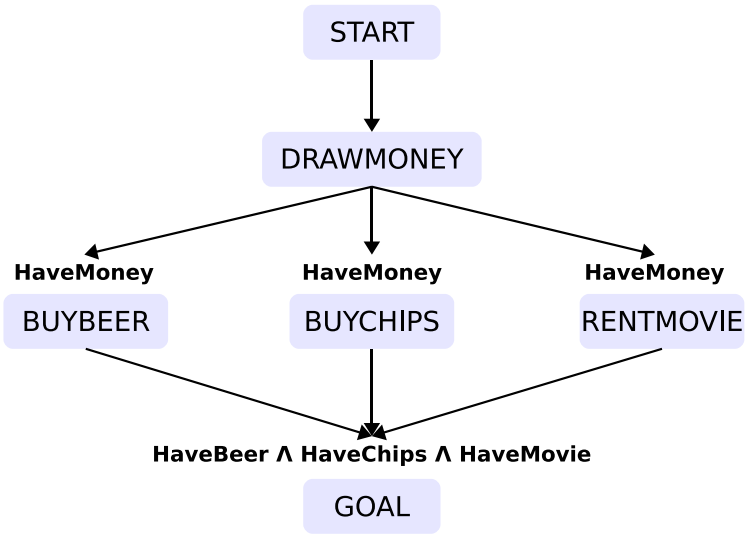


Figure 4.7: *Partial order plan for the lazy evening*

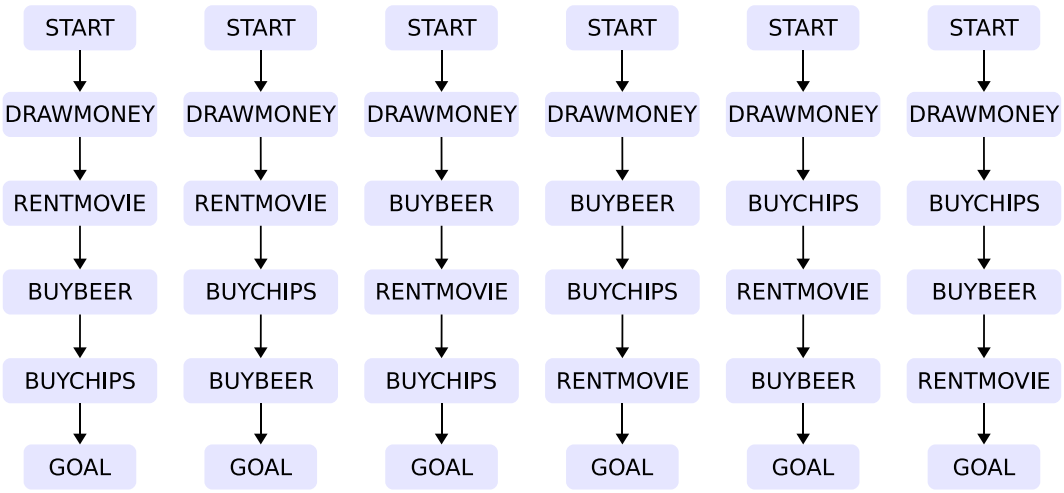


Figure 4.8: *Total order plans for the lazy evening*

entries are of the form $s_i < s_j$ where $s_i, s_j \in \mathcal{ST}$. This means that step s_i is performed before s_j.

Causal links are a method to keep track of past decisions. Causal links have the form $s_i \xrightarrow{c} s_j$ where s_i and s_j are steps of the plan. c is a proposition and an effect of s_i as well as a precondition of s_j, e.g. DRAWMONEY $\xrightarrow{HaveMoney}$ BUYBEER. It is read "s_i achieves c for s_j" and records e.g. that the purpose of DRAWMONEY is to achieve the precondition HaveMoney for BUYBEER. Causal links make it possible to detect *threats*. Let s_i, s_j, s_k be actions in $\mathcal{A}$, $s_i \xrightarrow{c} s_j$, and $\neg c$ is an effect of s_k. If not $s_k < s_i$ or $s_j < s_k$ then s_k threatens $s_i \xrightarrow{c} s_j$. If there was for example an action SPENDMONEY that could be executed between

DRAWMONEY and BUYBEER it threats DRAWMONEY $\overset{HaveMoney}{\longrightarrow}$ BUYBEER.

Variable binding constraints bind variables to either constants or reference them to other variables. In the blocks world example the variable x of the operator STACK(x,y) could be bound to block A, denoted as x=A. If only ground plan steps are used, i.e. plan steps without variables, as in the lazy evening example, then $\mathcal{B} = \emptyset$.

Other terms that can be derived from $\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL}, \mathcal{B} \rangle$ exist. $\mathcal{OC}$ is the set of open conditions. An open condition is a precondition c of a plan step s_j where there is no causal link $s_i \xrightarrow{c} s_j, s_i \in \mathcal{ST}$. $\mathcal{UCL}$ is the set of unsafe causal links, i.e. causal links that are threatened by another step. The *flaws* of a plan are the union of its open conditions and unsafe causal links: $\mathcal{F} = \mathcal{OC} \cup \mathcal{UCL}$.

A *solution* of a partial order planning problem is a *complete* and *consistent* plan. In a complete plan every precondition of every step is achieved by some other step. A consistent plan does not contain any contradictions in its ordering or binding constraints.

The initial partial plan, the plan at the begin of the search graph where no refinements have been made, is sometimes called the *null plan*. To improve simplicity and uniformity the null plan works with a little encoding trick to describe the initial planning problem. The null plan has the following structure:

- $\mathcal{ST} = \{\texttt{START}, \texttt{GOAL}\}$
- $\mathcal{O} = \{\texttt{START} < \texttt{GOAL}\}$
- $\mathcal{CL} = \emptyset$
- $\mathcal{B} = \emptyset$

START is a dummy step that has no precondition and its effects define the initial state of the system s_0.

START - effect: s_0

GOAL is a dummy step that has no effects but its preconditions are the propositions of the goal state s_G.

GOAL - precond: s_G

Plan-space planning was pioneered by Sacerdoti [Sac74, Sac90] who developed a planner called NOAH and reformulated the planning problem as a search through the space of plans. In 1977, Tate developed a system called NONLIN [Tat77] and introduced the concept of causal links.

TWEAK [Cha87] combined and distilled research in the field of partial order planners, formalised planning problems, and developed a sound and com-

plete generic planner. McAllester et al. [MR91] describe a simple, sound, complete, and systematic partial order planning algorithm for STRIPS planning, called SNLP.

The UCPOP planner [PW92] is a well-known plan space planner and introduced major improvements based on contributions of TWEAK and SNLP. It handles a subset of the ADL language. UCPOP is a regression planner and applies a search control strategy to decide which partial plan should be further refined.

REPOP [NK01] introduces several novel heuristic control techniques to improve the scalability of partial order planning algorithms. Its implementation is based on UCPOP.

VHPOP [YS03a] is a versatile heuristic partial order planner, loosely based on UCPOP. It was the best newcomer at the 3rd International Planning Competition in 2002. VHPOP comes with a set of flaw selection strategies to guide the search and permits to use them simultaneously to combine its strengths.

Graph-based planning

Graph-based planning uses a certain data structure for the planning process called *planning graph*. It can be used to obtain better heuristic estimates to guide the planning process. As illustrated in Figure 4.9, such heuristics can either be used together with a planner or a solution can directly be extracted from the planning graph, using a specialised algorithm.

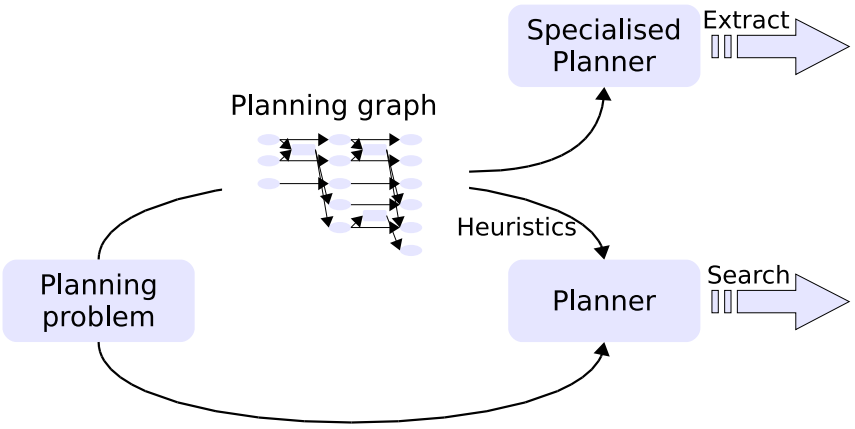


Figure 4.9: *Usage of planning graphs*

Planning graphs can only be applied to planning problems without variables. A planning graph consists of several levels that correspond to time steps in the plan (see Figure 4.10). Each level is layered and contains a state

layer and an action layer. The state layer of level 0, S_0, corresponds to the initial state s_0 of the planning problem. The action layer A_0 contains all actions whose preconditions are nodes in S_0. The basic idea is that the literals in S_t are those that *could* be true at time step t, A_t are those actions that *could* have their preconditions satisfied at step t. The action nodes are connected with its preconditions (the incoming edges) and its positive and negative effects (the outgoing edges).

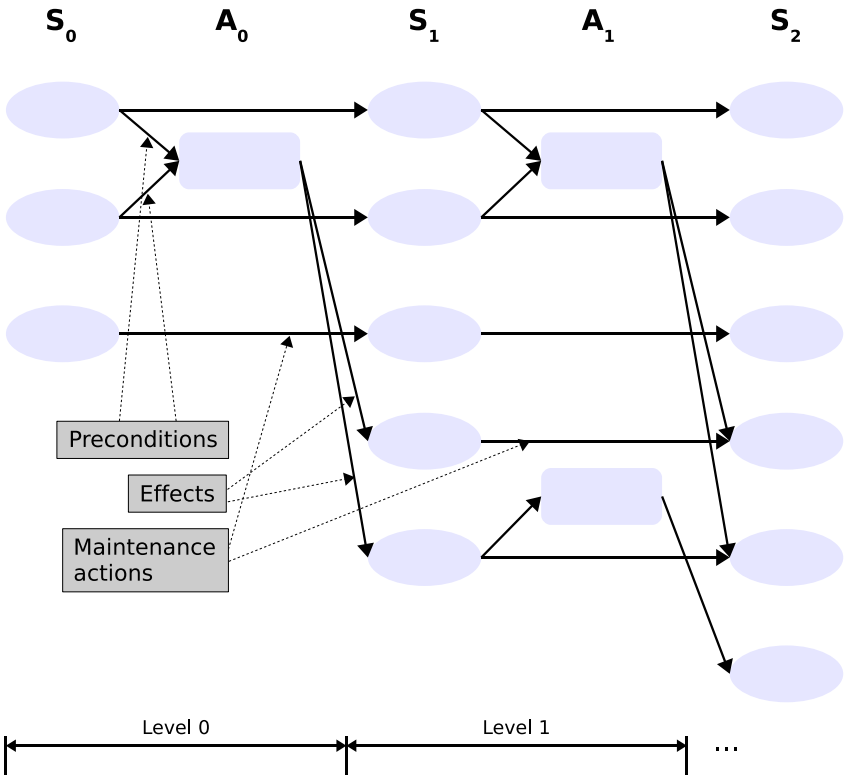
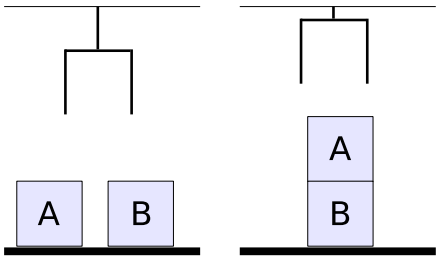


Figure 4.10: *A planning graph*

The planning graph does not take every possible interaction among actions into account. Therefore, the planning graph can be generated very efficiently. A planning graph can be seen as a relaxed problem for reachability analysis. It provides information on which states are reachable from s_0. However, in contrast to a full reachability analysis it only allows for a necessary criteria: If a state is reachable then it also appears in some node of the planning graph. Nevertheless, planning graphs can be used as a good estimate of how difficult it is to achieve a given state.

Figure 4.12 illustrates a part of a planning graph for the blocks world planning problem shown in Figure 4.11. The edges of a planning graph are directed, but for a better graphical representation the indication of the direction has been omitted. The states of S_0 represent the initial state s_0 of the planning problem. A_0 contains all actions whose preconditions are in



(a) Initial state (b) Goal state

Figure 4.11: *A planning problem within the blocks world*

S_0 and these actions are connected with their respective preconditions. S_1 contains all literals of S_0 plus literals that are contained in the effects of any action of A_0 and not contained in S_0. The actions of A_0 are also connected with their effects in S_1. Maintenance actions are dummy actions that link literals whose values do not change if no action is performed.

Not mentioned yet is another element of planning graphs: *mutexes*. They define pairs of elements that are mutual exclusive, i.e. cannot occur together. The action `PICKUP(A)` for instance is mutual exclusive with the maintenance actions `Armempty`. The mutexes are represented as red lines in Figure 4.12.

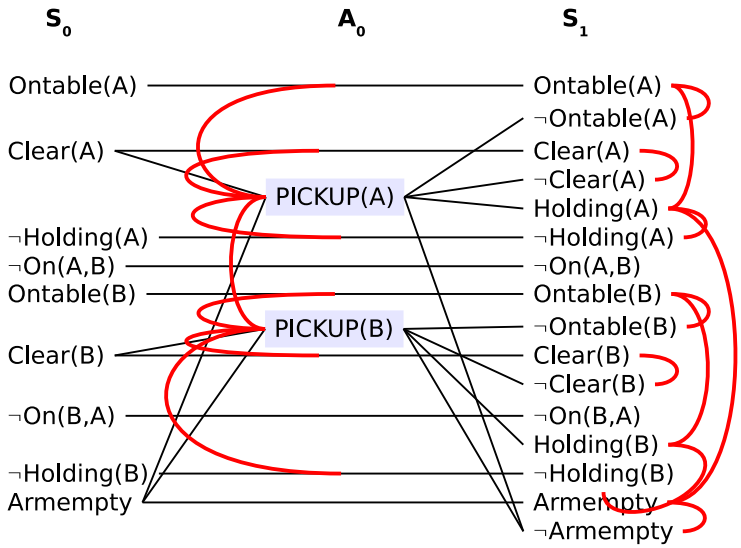


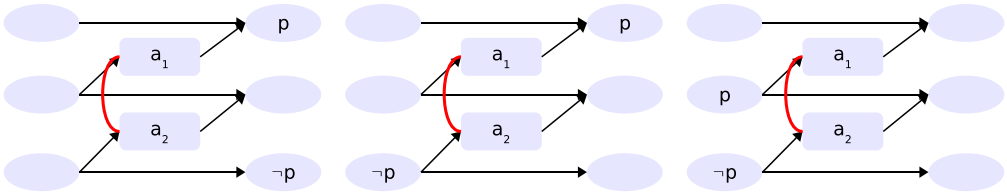
Figure 4.12: *Planning graph for the blocks world planning problem*

There are three different types of mutex relationships between two actions a_1 and a_2, see Figure 4.13:

Inconsistent effects: The action a_1 has an effect p while a_2 has an effect $\neg p$.

Interference: a_1 has an effect p while a_2 has a preconditions $\neg p$.

Competing needs: a_1 has a precondition p while a_2 has a precondition $\neg p$.



(a) Inconsistent effects (b) Interference (c) Competing needs

Figure 4.13: *Mutex conditions for actions*

For two literals p_1 and p_2 the criteria to be mutual exclusive is as follows: Either p_1 is the negation of p_2 or every action, also maintenance actions, that has p_1 as effect is mutex with every action that has p_2 as effect. This mutex relationship is called *inconsistent support*.

Further levels of the planning graph are constructed analogous. This comes to an end when two consecutive levels are identical. If two levels are once identical, then all the following levels are, too. The resulting graph provides information about which actions can be performed in which step together with the information which cannot be performed together. Similarly it provides information about which literals could occur and which combinations are impossible. This information can be used by planners to guide their search.

GRAPHPLAN [BF95] was the first planner that was based on a planning graph. It is an algorithm that directly extracts the solution from it. GRAPHPLAN operates in a loop with two alternating steps: planning graph expansion and solution extraction. The graph is expanded until a state level S_x is reached which includes all goal literals without mutex relations between them. This is a necessary condition, i.e. the graph could contain a solution. In this stage the algorithm tries to extract a plan within the solution extraction phase. If this fails the graph is further expanded. This loop terminates if either a solution is found or it turns out that there is no solution.

SGP [WAS98] is a descendant of GRAPHPLAN that solves contingent planning problems, i.e. generation of a plan whose course of action depends on information based on sensing actions. Therefore SGP modifies the graph expansion phase and incorporates a conditioning threat resolution method into the solution extraction phase.

TGP [SW99] is a planner for temporal planning. TGP adopts an extension of the STRIPS language allowing to assign starting times and durations to actions. The planning algorithm generates a temporal planning graph to find a solution for the planning problem.

Koehler et al. [KNHD97] present an early version of their IPP planner, in which they extend GRAPHPLAN to a subset of ADL. It allows conditional and universally quantified effects in operations while most interesting prop-

erties of GRAPHPLAN are preserved. In later work IPP has been extended by a technique to remove irrelevant operators and facts from planning problems as well as a goal agenda manager to order subgoals and plan for subproblems [KH00].

STAN [FL01], another planner based on GRAPHPLAN, introduces an efficient structure to represent the planning graph. Thus it reduces the costs for graph construction. Furthermore, it uses TIM [FL98, LF00], a domain analysis tool, to improve the planning process.

Planning as satisfiability

The traditional way of planning based on logic is deduction [Gre69, Ros81]. In this case planning is proving the theorem that the initial state together with axioms that describe the effects of the actions and some sequence of actions imply the goal conditions. However, this technique is not very efficient. Another logical planning approach is to test the satisfiability of logical expressions instead of theorem proving. Basically, this refers to finding a model that satisfies a logical sentence like this [RN95]:

$$\texttt{initial state} \wedge \texttt{all possible action descriptions} \wedge \texttt{goal}$$

This sentence contains the initial state s_0 of the planning problem, a goal state s_G, and logical expressions corresponding to every possible action. A model that satisfies the sentence will assign true to the actions that are part of a correct plan and false to the others. The sentence is unsatisfiable if no solution of the planning problem exists.

Planning as satisfiability was first proposed by Kautz and Selman [KS92, KS96] and used in their SATPLAN planner. The encoding in a satisfiability problem was hand-coded as they had difficulties to automatically derive it from STRIPS.

Ernst et al. [EMW97] developed an automatic "compiler" to generate satisfiability problems from planning problems in PDDL representation.

BLACKBOX [KS98, KS99] is a planner that combines SATPLAN and GRAPHPLAN. It takes planning problems specified in STRIPS as input and converts them to a planning graph of a certain length. The plan graph is then converted to a special logic representation. The resulting logic formula is then simplified by a general simplification algorithm. This formula is then solved by a satisfiability planner. If a model could be found it is converted to a plan, otherwise the process is repeated with a planning graph of an incremented size.

Further satisfiability planners are e.g. DPLL [Lib00], WALKSAT [SKC95], and LPSAT [WW99].

With the given survey of planning techniques it is tried to cover some important aspects. For more information about automated planning it is referred to [GNT04].

In the next section a failure recovery engine is introduced which is able to manage distributed systems.

4.4 Failure recovery engine

The failure recovery process can be seen as an instance of an Observer/Controller architecture [RMB+06] in the terminology of Organic Computing (OC) or the MAPE [SPTU05] control loop of Autonomic Computing (AC) systems. The MAPE control loop consists of the four stages monitor, analyse, plan, and execute.

The failure recovery engine works in a distributed way where one instance runs on every node of a distributed system. The user/administrator defines objectives and the engine is responsible for their compliance. Thus, users can specify the desired system behaviour but do not need to deal with its implementation. Monitoring and analysing provides information about the system while planning and execution yields decision-making and action-taking.

The planning capabilities of the recovery engine are based on the POP algorithm which is introduced in the next section.

4.4.1 Pop algorithm

The POP algorithm [Wel94] shown in Algorithm 13 is a partial order plan space planner (see Section "Plan-space planning" in 4.3.2).

Recall that partial order plans can be represented as a quadruple: $\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL}, \mathcal{B} \rangle$ where

- $\mathcal{ST}$ is a set of steps,
- $\mathcal{O}$ is a set of ordering constraints,
- $\mathcal{CL}$ is a set of causal links, and
- $\mathcal{B}$ is a set of variable binding constraints.

For the sake of simplicity, in the POP algorithm (Algorithm 13), the usage and binding of variables is omitted. In this case a plan can be represented as $\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL} \rangle$. $\mathcal{A}$ is the set of available actions of the system. A set named *agenda* is used to keep track of the open conditions $\mathcal{OC}$. Each item of *agenda*

Algorithm 13 The POP algorithm

function POP($\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL} \rangle, agenda, \mathcal{A}$)

[1. Termination:]
if *agenda* is empty **then return** $\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL} \rangle$
end if

[2. Select flaw:]
Choose an item $\langle c, a_{need} \rangle$ from *agenda*

[3. Select action:]
Choose an action a_{add} that has c as effect;
either from the steps of the plan $\mathcal{ST}$ or
a new action from the set of available actions $\mathcal{A}$.
if no such actions exists **then return** 'failure'
end if

[4. Refine plan:]
Add a_{add}, an ordering constraint, and a causal link.
$\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL} \rangle :=$
$\langle \mathcal{ST} \cup \{a_{add}\}, \mathcal{O} \cup \{a_{add} < a_{need}\}, \mathcal{CL} \cup \{a_{add} \xrightarrow{c} a_{need}\} \rangle$.

[5. Update *agenda***:]**
Remove $\langle c, a_{need} \rangle$ from *agenda*:
$agenda := agenda \setminus \langle c, a_{need} \rangle$.
Then add an entry $\langle c_i, a_{add} \rangle$ to *agenda* for all preconditions c_i of a_{add}.

[6. Resolve threats:]
For each action a_{threat} that threatens a link
$a_i \xrightarrow{c} a_j$ in $\mathcal{CL}$ choose either
(a) Promotion: Add $a_{threat} < a_i$ to $\mathcal{O}$
(b) Demotion: Add $a_j < a_{threat}$ to $\mathcal{O}$

[7. Recursive call:]
POP($\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL} \rangle, agenda, \mathcal{A}$)

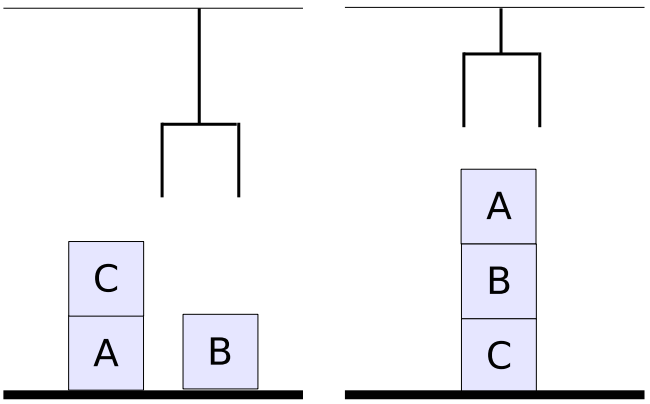
end function

has the form $\langle c, a_i \rangle$ where c is a precondition of a_i. The initial call of POP is performed with the parameters:

- The null plan representing the planning problem: $\langle \{\texttt{START}, \texttt{GOAL}\}, \{\texttt{START} < \texttt{GOAL}\}, \emptyset \rangle$.

- *agenda* consists of the subgoals of the initial planning problem, i.e. $agenda := \{\langle c_1, \texttt{GOAL}\rangle, \ldots, \langle c_n, \texttt{GOAL}\rangle\}$, where $c_1, \ldots, c_n$ are the preconditions of `GOAL`.
- The available actions of the planning problem: $\mathcal{A}$.

The POP algorithm terminates if there are no more flaws. In every call the algorithm selects one flaw and tries to achieve it. To do so, the algorithm chooses a refinement, normally either to use an existing step from $\mathcal{ST}$ or to add a newly instantiated one from $\mathcal{A}$. If no action exists that could achieve the flaw then a failure is returned. Assuming that the action a_{add} is chosen to achieve subgoal $\langle c, a_{need}\rangle$, i.e. a_{add} has c as an effect. Then a_{add} is added to the set of steps $\mathcal{ST}$ if a_{add} is a newly instantiated action. If a_{add} is an already existing step nothing needs to be done at this point. In either case $a_{add} < a_{need}$ is added to $\mathcal{O}$ and $a_{add} \xrightarrow{c} a_{need}$ is added to $\mathcal{CL}$ to commit that a_{add} is performed before a_{need} and a_{add} achieves c for a_{need}. As now the agenda item $\langle c, a_{need}\rangle$ is achieved it can be removed. In return, for all preconditions c_i of a_{add} an entry $\langle c_i, a_{add}\rangle$ is added to the agenda to declare that these preconditions have to be achieved in a subsequent step. The addition of $a_{add} \xrightarrow{c} a_{need}$ to $\mathcal{CL}$ could cause threats. If there is for example a causal link $a_i \xrightarrow{\neg c} a_j$ and not $a_{need} < a_i$ or $a_j < a_{need}$ then a_{add} threatens $a_i \xrightarrow{\neg c} a_j$. To resolve such a threat either promotion or demotion can be used. Promotion in this case is to add $a_{need} < a_i$, demotion to add $a_j < a_{add}$ to the ordering constraints $\mathcal{O}$.



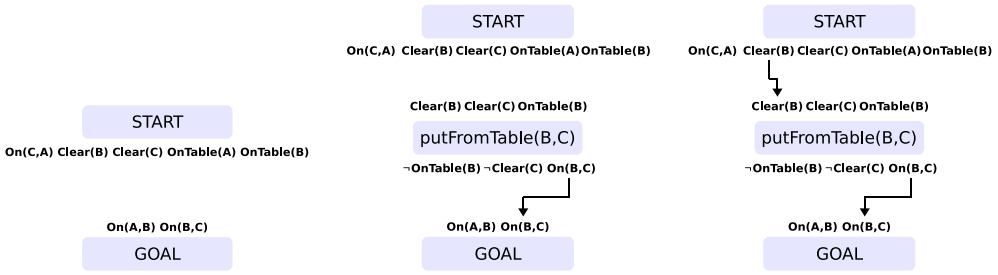
(a) Initial state (b) Goal state

Figure 4.14: *The "Sussman anomaly" planning problem*

To demonstrate the functionality of the POP algorithm it is applied to an example planning problem: the so-called "Sussman anomaly" [Sus75]. Planners that are based on the assumption that subgoals are independent and can be sequentially achieved in an arbitrary order have problems to solve this problem. Figure 4.14 illustrates the Sussman anomaly in the blocks world domain. The problem has two subgoals: `On(A,B)` and `On(B,C)`. If it is first tried to achieve `On(A,B)`, then it is obvious to put `C` on the table and `A`

on B afterwards. However, now it is impossible to achieve On(B,C) without destroying On(A,B). Trying to achieve On(B,C) first results in putting B on C. As A remains on the table under C, in this case it is impossible to achieve On(A,B) without destroying On(B,C). The root of the problem is that the subgoals interfere with each other. Also Weld [Wel94] uses the Sussman anomaly to explain the POP algorithm.

Figure 4.15(a) shows the null plan of the Sussman anomaly. This is used as argument for the call of the POP procedure. START is a dummy step that has no precondition and its effects define the initial state of the problem, GOAL is a dummy step that has no effects but its preconditions represent the goal state. For the sake of simplicity two new actions are introduced that are compositions of the already defined ones of the blocks world: PutFromTable(X,Y) takes a block X that is on the table and puts it on Y. It is a composition of Pickup(X) and Stack(X,Y). The second action putOnTable(X,Y) takes C that is currently on Y and puts it on the table. This combines Unstack(X,Y) and Putdown(X). The set $\mathcal{A}$ contains all the available actions of the planning domain, in this case Pickup(X), Putdown(X), Stack(X,Y), Unstack(X), PutFromTable(X,Y), and putOnTable(X,Y). At the call of the POP algorithm the agenda consists of the two elements ⟨On(A,B), GOAL⟩ and ⟨On(B,C), GOAL⟩.



(a) Initial plan (b) After first call (c) After second call

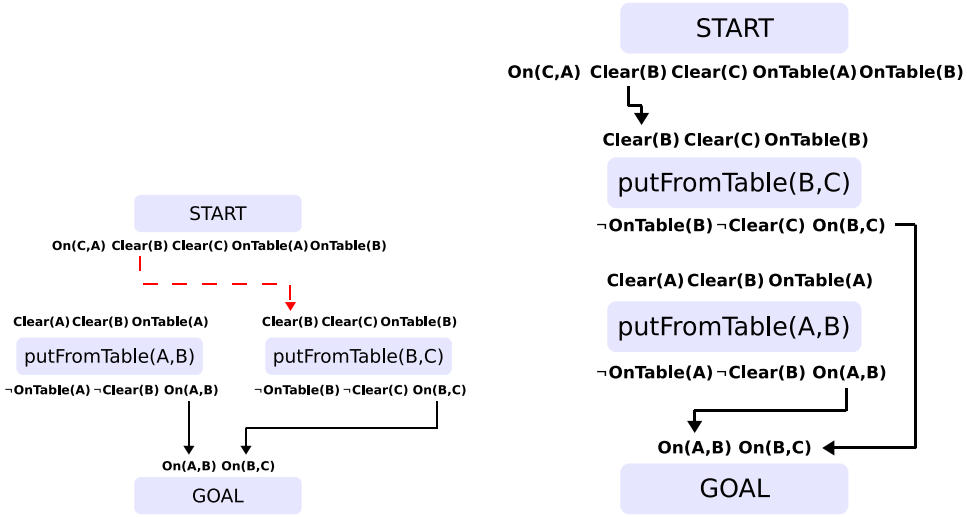
Figure 4.15: *Partial plans for the "Sussman anomaly"*

As the list *agenda* is not empty the algorithm (see Algorithm 13) proceeds to step 2 and selects one item of the agenda as flaw to achieve next. The choice of the flaw has no influence on the completeness of the planning process, but it has an influence on the performance of the search. It falls into the scope of planning heuristics to select opportune flaws. Let us assume the algorithm chooses the flaw ⟨On(B,C), GOAL⟩ first. In the next step the algorithm selects an action either from $\mathcal{ST}$ or from $\mathcal{A}$ that has On(B,C) as an effect. In this example it selects PutFromTable(B,C). According to step 4 the algorithm adds PutFromTable(B,C) to $\mathcal{ST}$, PutFromTable(B,C) $<$ GOAL to the ordering constraints $\mathcal{O}$, and PutFromTable(B,C) $\xrightarrow{On(B,C)}$ GOAL to the set of causal links $\mathcal{CL}$. Then, the agenda is updated: ⟨On(A,B), GOAL⟩ is now achieved by a causal link and can be removed. In return,

it adds entries for all preconditions of `PutFromTable(B,C)` to the agenda, i.e. $\langle$`Clear(B), PutFromTable(B,C)`$\rangle$, $\langle$`Clear(C), PutFromTable(B,C)`$\rangle$, and $\langle$`OnTable(B), PutFromTable(B,C)`$\rangle$. Since there is only one causal link a threat cannot occur. As final step of this first call, the POP procedure is recursively invoked. The corresponding partial plan at the end of the first call is shown in Figure 4.15(b) where the arrow denotes the causal link.

In the second run of the POP procedure it is assumed that the flaw $\langle$`Clear(B), PutFromTable(B,C)`$\rangle$ is chosen and the existing step `START` is selected to achieve it. Figure 4.15(c) illustrates the resulting partial plan.

Suppose that in the third invocation of the POP procedure the planner selects the flaw $\langle$`On(A,B), GOAL`$\rangle$ and the newly instantiated action `PutFromTable(A,B)` from $\mathcal{A}$ to achieve it. In this case the planner recognises a threat as the newly introduced step has an effect `PutFromTable(A,B)` ¬`Clear(B)` and could be executed before the `PutFromTable(B,C)` action. Therefore it threatens the causal link `START` $\xrightarrow{Clear(B)}$ `putFromTable(B,C)` as depicted in Figure 4.16(a). To resolve this threat, the POP algorithm chooses *demotion*, i.e. adds `PutFromTable(B,C)` $<$ `PutFromTable(A,B)` to the ordering constraints $\mathcal{O}$. The partial plan after the resolution of the threat is shown in Figure 4.16(b).



(a) Partial plan with threat (b) Partial plan with resolved threat

Figure 4.16: *Partial plans for the "Sussman anomaly"*

The POP procedure is recursively called as long as the agenda is not empty and finally results in the plan that is illustrated in Figure 4.17. In this plan all the preconditions are achieved by a causal link without any threats.

The proposed recovery engine is based on the POP algorithm presented in this section, but with several extensions. In the following the developed failure recovery concepts are presented. In Section 4.4.2 the planning language

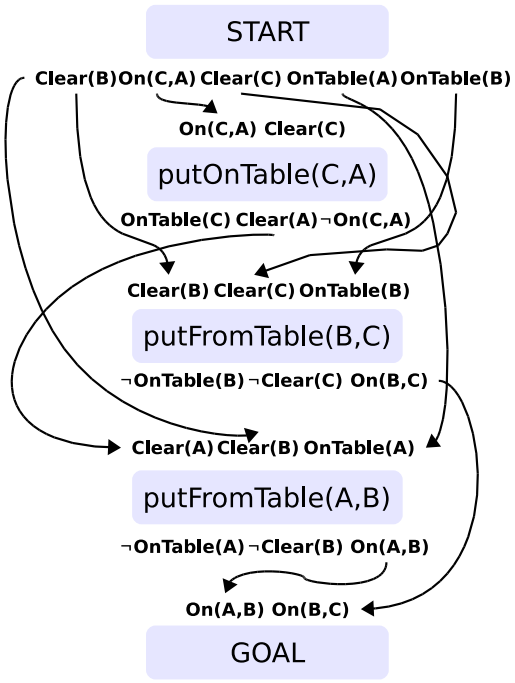


Figure 4.17: *Final plan for the "Sussman anomaly"*

used to specify e.g. the system objectives is introduced. Section 4.4.3 discusses the functionality of the failure recovery engine in detail. Section 4.4.4 introduces further extensions and improvements of the failure recovery process.

4.4.2 Planning language

To specify the system objectives, the actions nodes are able perform, and to describe the current condition of the system a planning language is used. The planning language follows the syntax of the Planning Domain Definition Language PDDL [McD98], presented in 4.3.1. A PDDL definition consists of a domain and a problem definition. In this section the general input language for the developed planner is introduced.

Domain definition

The domain definition contains the definitions for types, predicates and for the available actions.

```
(:types TYPE_1 ... TYPE_N)
(:predicates (PREDICATE_1_NAME [ARG_1 ... ARG_N])
             (PREDICATE_2_NAME ...
     ...)

(:action ACTION_1_NAME
  [:parameters (PARAM_1 ... PARAM_N)]
  [:precondition PRECOND_FORMULA]
  [:effect EFFECT_FORMULA]
 )

(:action ACTION_2_NAME
  ...)

...
```

Elements inside of angled parentheses are optional. A type like `TYPE_1` is defined by a name and a supertype: `TYPE_1_NAME [- SUPERTYPE_TYPE_1]`. Types are organised hierarchically where every type has exactly one supertype. If no supertype is given the standard supertype `object` is implicitly set. The arguments of predicates and the parameters of actions are typed variables. Variables are indicated by question marks. The argument `ARG1` is defined by `?ARG_1_NAME - TYPE_ARG_1`.

In the current implementation of the planning engine, a precondition formula `PRECOND_FORMULA` is any formula of the predicate calculus which is in prenex normal form, i.e. `PRECOND_FORMULA`$= Q_1 y_1 Q_2 y_2 \dots Q_k y_k F$, where $k \geq 0$ and $Q_1, \dots, Q_k \in \{\forall, \exists\}$ and no further quantors in F. Other restrictions are currently that in F negation occurs only immediately above elementary propositions, and $\neg, \wedge$ are the only allowed connectives.

An effect formula `EFFECT_FORMULA` is of the same form as a precondition formula, but as a further restriction no existential quantifiers are allowed.

Problem definition

The problem definition contains the objects present in the problem instance, the current state of the system, and its goal.

```
(:objects OBJ1 OBJ2 ... OBJ_N)
(:init ATOM1 ATOM2 ... ATOM_N)
(:goal PRECOND_FORMULA)
```

The system's state description consists of a list of all atoms true in the initial state. All other atoms are regarded as false. The system's goal description has the same form as an action precondition.

To clarify the approach a simple scenario is discussed, a small automated production cell, introduced by Güdemann et al. [GOR06]. It consists of three robots and two carts. Each robot has three tools which can be switched during runtime: (1) a drill to drill holes into workpieces, (2) an inserter to insert a screw into a drilled hole, and (3) a screw driver to tighten an inserted screw. The carts transport workpieces from one robot to another. All workpieces must be processed in the order drill, insert, and tighten. Figure 4.18 illustrates this scenario. The arrows represent the flow of the workpieces, the tools below the robots are the available tools for that robot, tools marked with a dot indicate the robot's tasks, i.e. the processing steps it is responsible to execute.

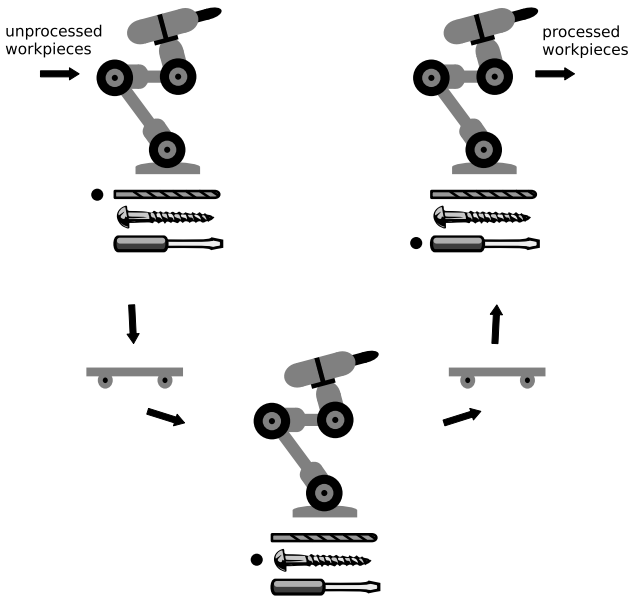


Figure 4.18: *Production cell*

As a start it is shown how this system can be modelled in order to be managed by a classical automated planning approach which requires an omniscient central instance performing the planning.

Below the domain definition for the production cell scenario is given.

```
(:types robot tool cart - object)

(:predicates (having ?r - robot ?t - tool)
             (using ?r - robot ?t - tool)
             (transporting ?c - cart ?from ?to - tool)
)

(:action startTool
   :parameters   (?r - robot  ?t - tool)
   :precondition (and  (having ?r ?t) (not (using ?r ?t)))
   :effect       (and  (using ?r ?t))
)

(:action stopTool
   :parameters   (?r - robot  ?t - tool)
   :precondition (and  (using ?r ?t))
   :effect       (and  (not (using ?r ?t)))
)

(:action startTransport
   :parameters   (?c - cart  ?from ?to - tool)
   :precondition ( )
   :effect       (and  (transporting c1 ?from ?to))

(:action stopTransport
   :parameters   (?c - cart  ?from ?to - tool)
   :precondition (and  (transporting ?c ?from ?to))
   :effect       (and  (not (transporting ?c ?from ?to)))
)
```

Thus, robots can start and stop tools. In order to start a tool `?t` the corresponding tool must be available for the robot `(having r1 ?t)`. The effect of the action `startTool` is that the robot is using the tool afterwards `(using r1 ?t)`. A robot can also use two tools simultaneously. This is especially important if another robot completely fails and the remaining ones have to take over its tasks. The action `stopTool` can be seen as the complementary action to `startTool` and stops using a tool. Carts transport workpieces from one robot which is currently using a certain tool to a robot using another tool. Similar to robots they also start and stop these tasks.

As goal only a subcondition of the condition necessary for ensuring the full functionality of the production cell is presented. For the purpose of explain-

ing the functionality of the planning engine this is sufficient and easier to comprehend.

```
(:objects r1 r2 r3 - robot
          drill inserter screw_driver - tool
          c1 c2 - cart
)

(:init
 (having r1 drill)(having r1 inserter)(having r1 screw_driver)
 (having r2 drill)(having r2 inserter)(having r2 screw_driver)
 (having r3 drill)(having r2 inserter)(having r3 screw_driver)

 (using r1 drill)  (using r2 inserter) (using r3 screw_driver)

 (transporting c1 drill inserter)
 (transporting c2 inserter screw_driver)
)

(:goal (forall (?t - tool) (exists (?r - robot) (using ?r ?t)))
)
```

It is supposed that in the current system condition all three robots have the tools `drill`, `inserter`, and `screw_driver`, and `r1` is currently using the `drill`, `r2` the `inserter`, and `r3` the `screw_driver`. The cart `c1` is transporting workpieces from the robot using the `drill` to the robot using the `inserter` and `c2` is transporting workpieces from the robot using the `inserter` to the robot using the `screw_driver`. In this example only the predicates which are true are cited, implicitly also e.g. `(not (using r1 inserter))` is holding.

The objective considered for explanation purposes is that for each tool there must be one robot which is using that tool.

The next section presents the capabilities of the planning engine to work in distributed domains with no central omniscient instance.

4.4.3 Failure recovery process

It is assumed that the self-healing approach is applied to a distributed system consisting of m nodes $n_1, \ldots, n_m$. The nodes perceive their environment with sensors. The term *sensor* stands for any facility that contributes to gather information about external or internal states. Typically, nodes only have sensors for some facts, but do not have information about the whole

system. All nodes are running an instance of the distributed planning engine.

The current state of the system is expressed by predicates. Let $\mathscr{S}$ be the set of existing sensors, and $\mathscr{P}$ be the set of existing predicates. A function $\mu : \mathscr{S} \rightarrow \mathscr{P}$ maps a sensor to its corresponding predicate. The function μ is injective; in the untypical case that a node has a sensor for each predicate, which represents a complete view on the whole system, it is also surjective. The set $\mathscr{P}_n$ stands for all predicates, node n has a sensor to determine its value, i.e. $\mathscr{P}_n \subseteq \mathscr{P}$ in more detail $\mathscr{P}_n = \{p \in \mathscr{P} \mid \exists s \in \mathscr{S} : \mu(s) = p\}$.

Nodes are only able to perceive parts of the system status, but it is assumed that there is no predicate which cannot be perceived by any node. This means in a system with the nodes $n_1, \ldots, n_m$, the union of the perceivable predicates of all nodes results in the set of all predicates: $\mathscr{P}_{n_1} \cup \ldots \cup \mathscr{P}_{n_m} = \mathscr{P}$.

All entities in the distributed system constantly monitor their environment using their sensors. If one entity observes a violation of an objective it initiates a recovery process. This entity now serves as coordinator for a distributed planning process with the goal to recover the system to a state in line with the system's objectives. The coordinator manages an agenda with open conditions that have to be addressed in order to recover the system. The open conditions are announced, while all entities communicate their possible contributions to resolve an open condition to the coordinator. Thus, the coordinator is able to generate a plan where the abilities of other entities are included. The result is a parallel executable plan that recovers the system from the unwanted state. Using the MAPE architecture, the failure recovery process for each node can be expressed as follows:

Monitor: Relevant environmental parameters are monitored using the available sensors.

Analyse: Check whether the system is consistent with objectives specified in PDDL.

Plan: The node which first discovers an inconsistency initiates and coordinates a distributed planning process.

Execute: The found plan is executed in a distributed way.

The analyse, plan, and execution phases of the failure recovery engine are the most interesting ones and explained in the following sections. Before, the information restrictions of planning in distributed environments in com-

parison to classical planning are discussed.

Types: The available types are known due to information exchange of the nodes.

Predicates: Each node only knows about the existence of predicates necessary to describe its actions, the goal, and the initial state. However, the current value of a predicate is only known if the node has a corresponding sensor.

Actions: Each node only knows the actions it is able to perform itself.

Objects: The available objects are known due to information exchange of the nodes.

Init: The initial state reflects the system's state in this work. Each node only has information about the parts of the system covered by its sensors.

Goal: Any node knows the complete goal description for the system. This goal description may be broadcast into the network.

The information restrictions are exemplified using the production cell scenario introduced above from the view of the robot `r1`.

```
(:types robot tool cart - object)

(:predicates (having ?r - robot ?t - tool)
             (using ?r - robot ?t - tool)
)

(:action startTool
   :parameters   (r1  ?t - tool)
   :precondition (and  (having r1 ?t) (not (using r1 ?t)))
   :effect       (and  (using r1 ?t))
)

(:action stopTool
   :parameters   (r1  ?t - tool)
   :precondition (and  (using r1 ?t))
   :effect       (and  (not (using r1 ?t)))
)
```

The existing types, `robot`, `tool`, and `cart`, are known to robot `r1`, as it exchanges the information about the available types with the other nodes. Robot `r1` is aware of the predicates `having` and `using` since these predicates are used to describe its actions and the system's goal. The list of actions consists only of the actions `r1` is able to perform.

```
(:objects r1 r2 r3 - robot
          drill inserter screw_driver - tool
)

(:init
 (having r1 drill)(having r1 inserter)(having r1 screw_driver)
 (using r1 drill) (not (using r1 inserter))
 (not (using r1 screw_driver))
)

(:goal (forall (?t - tool) (exists (?r - robot) (using ?r ?t)))
)
```

Based on message exchange, robot `r1` is aware of the existing objects of type `robot` and `tool`. In fact, information about all objects of a type is only necessary if a `forall` expression is used which quantifies over this type. In the `:init` section only the values of predicates are known the robot has a sensor for. In this case the robots know about their own tools. Different from the description of the system's state using an omniscient planner, not only the predicates which are true are cited but also the predicates whose value is false are explicitly stated. Predicates which are not stated have the new status *unknown*.

Analyse: consistency check

Any node in the distributed system is constantly checking whether the system is in line with the stated objectives. These are expressed in a precondition formula in the `:goal` section. First the distributed consistency check is discussed for a goal formula without quantifiers. Thus, the goal precondition formula `G` consists just of a conjunction of positive and negative predicates. Two variants for the analysis whether a system is in a valid state exist: (1) Each node inspects the system to be in line with the full list of goal predicates which is reasonable for smaller systems and (2) the goal is divided into subgoals and the task of consistency checks for these subgoals is assigned to different nodes.

Assuming the goal has the form $G = g_1 \wedge \ldots \wedge g_k$, where g_i $(1 \leq i \leq k)$ is a positive or negative predicate. For a consistency check the elements $g_1, \ldots, g_n$ need to be assigned to the nodes $n_1, \ldots, n_m$ of the system . In the case (1) each node monitors the value for each element of G. If a node does not have a sensor to directly diagnose the value of the corresponding predicate, it needs to ask nodes which have such a sensor. If the monitoring of the goal predicates is divided and each node is only responsible for parts of G, it is of course necessary for the correct functioning that all goal predicates

have at least one node which is checking it. For an efficient consistency check the distribution of subgoals should consider to assign predicates to nodes which can directly observe them.

To be able to divide and distribute goals which contain a quantifier, the planning engine eliminates all universal quantifiers. This can easily be performed as the quantified variables are typed and their possible values - the objects - are finite. Thus, the goal G is transformed to the form $G = \exists y_1 \ldots \exists y_i G$ instead of $G = Q_1 y_1 \ldots Q_k y_i F$, where $Q_1, \ldots, Q_k \in \{\forall, \exists\}$. F then has the form $F = g_1 \wedge \ldots \wedge g_k$, where g_i $(1 \leq i \leq k)$. This can be transformed to $G = \exists y_1 \ldots \exists y_i g_1 \wedge \ldots \wedge \exists y_1 \ldots \exists y_i g_k$. These k subgoals can then again be distributed to different nodes to perform consistency checks. The consistency check for existentially quantified predicates demands that one actual instance of a corresponding predicate needs to be accounted.

In the robot cell example, first the universal quantifier of the goal `(forall (?t - tool) (exists (?r - robot) (using ?r ?t)))` is eliminated which results in a conjunction of `(exists (?r - robot) (using ?r drill))`, `(exists (?r - robot) (using ?r inserter))`, and `(exists (?r - robot) (using ?r screw_driver))`. These subgoals are distributed to the nodes of the production cell, i.e. the robots `r1`, `r2`, `r3`, and the carts `c1`, `c2`. To check e.g. the subgoal `(exists (?r - robot) (using ?r drill))`, a node verifies whether at least on of the predicates `(using r1 drill))`, `(using r2 drill))`, and `(using r3 drill))` is true.

If any node finds an inconsistent subgoal, the planning phase is supposed to resolve that problem and to transform the system into a state which is in line with the objectives.

Plan: distributed planning

If one entity observes a violation of an objective it initiates a reconfiguration process. This entity now serves as coordinator for a distributed planning process which is performed with a distributed variant of the POP algorithm, called DPOP. It has been developed in the course of this work and is shown in Algorithm 14. The red colour indicates the parts which differ substantially from the central POP algorithm.

In step 2, the POP as well as the DPOP algorithm select a flaw which needs to be achieved. The central algorithm selects an action to do so from its local set of available actions. DPOP, however, broadcasts the selected flaw to all nodes. Note that each node runs an instance of the planning engine implementing the DPOP algorithm. Each node has a list of actions containing the regular actions it can perform as well as the two dummy actions `INIT` and `GOAL`. Like the POP algorithm, DPOP represents the system's goals as a `GOAL`

Algorithm 14 The DPOP algorithm

function DPOP($\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL} \rangle, agenda$)

[1. Termination:]
if *agenda* is empty **then return** $\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL} \rangle$
end if

[2. Select flaw:]
Choose an item $\langle c, a_{need} \rangle$ from *agenda*

[3. Broadcast flaw:]
The chosen flaw $\langle c, a_{need} \rangle$ is broadcast to all nodes.
For uniformity the broadcast is also received by the coordinator itself which also answers it.

[4. Receive actions:]
All nodes receiving a flaw $\langle c, a_{need} \rangle$ send their actions which have c as effect from their set of available actions or from existing steps to the coordinator. Each node has at least one existing step *INIT* which has the effects to introduce the initial predicates, i.e. the predicates of the `:init` section.

[5. Select action:]
Choose an action from the received action offers.
if no actions have been received **then return** 'failure'
end if

[6. Refine plan:]
Add a_{add}, an ordering constraint, and a causal link.
$\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL} \rangle :=$
$\langle \mathcal{ST} \cup \{a_{add}\}, \mathcal{O} \cup \{a_{add} < a_{need}\}, \mathcal{CL} \cup \{a_{add} \xrightarrow{c} a_{need}\} \rangle$.

[7. Update *agenda***:]**
Remove $\langle c, a_{need} \rangle$ from *agenda*:
$agenda := agenda \setminus \langle c, a_{need} \rangle$.
Then add an entry $\langle c_i, a_{add} \rangle$ to *agenda* for all preconditions c_i of a_{add}.

[8. Resolve threats:]
For each action a_{threat} that threatens a link $a_i \xrightarrow{c} a_j$ in $\mathcal{CL}$ choose either
(a) Promotion: Add $a_{threat} < a_i$ to $\mathcal{O}$
(b) Demotion: Add $a_j < a_{threat}$ to $\mathcal{O}$

[9. Recursive call:]
DPOP($\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL} \rangle, agenda$)

end function

step. However, the INIT step of DPOP contains only the known predicates. If a node receives a broadcast request containing a flaw, it uses the planning engine to find all actions which achieve that flaw. This set of actions is then sent to the coordinator which can conduct the further planning with these actions. Coming back to the production example, Robot r1 for instance has the following four actions.

```
(:action startTool
    :parameters   (r1  ?t - tool)
    :precondition (and  (having r1 ?t) (not (using r1 ?t)))
    :effect       (and  (using r1 ?t))
 )

 (:action stopTool
    :parameters   (r1  ?t - tool)
    :precondition (and  (using r1 ?t))
    :effect       (and  (not (using r1 ?t)))
 )

 (:action INIT
    :parameters
    :precondition
    :effect      (and (having r1 drill) (having r1 inserter)
                      (having r1 screw_driver) (using r1 drill)
                      (not (using r1 inserter))
                      (not (using r1 screw_driver)))
 )

 (:action GOAL
    :parameters
    :precondition (forall (?t - tool)
                  (exists (?r - robot) (using ?r ?t)))
    :effect
 )
```

Consider the production cell as presented in Figure 4.18, but robot r3 has failed and is now completely unavailable. The goals are: `(exists (?r - robot) (using ?r drill))`,`(exists (?r - robot)(using ?r inserter))`, and `(exists (?r - robot) (using ?r screw_driver))`. As robot r3 failed while using the tool screw_driver, the last goal is false. This is recognised in the analyse phase and a distributed planning process is initiated to recover the system. Assuming that robot r1 becomes the coordinator for the planning process an optimal planning sequence looks as follows:

For each recursive run of the DPOP function (see Algorithm 14), the agenda and the set of causal links ($\mathcal{CL}$) is stated. It is shown which flaw the coordinator is selecting and broadcasting. Furthermore, it can be seen which actions are received by the coordinator as a result of the broadcast and which action it selects.

1. Call

$agenda = \{\langle D, GOAL\rangle, \langle I, GOAL\rangle, \langle S, GOAL\rangle\}$
$\mathcal{CL} = \emptyset$

`r1` broadcasts D and receives:

from `r1`: `INIT`, `startTool(r1,D)` from `r2`: `startTool(r2,D)`

`r1` selects `INIT`

2. Call

$agenda = \{\langle I, GOAL\rangle, \langle S, GOAL\rangle\}$
$\mathcal{CL} = \{INIT \xrightarrow{D} GOAL\}$

`r1` broadcasts I and receives:

from `r1`: `startTool(r1,I)` from `r2`: `INIT`, `startTool(r2,I)`

`r1` selects `INIT`

3. Call

$agenda = \{\langle S, GOAL\rangle\}$
$\mathcal{CL} = \{INIT \xrightarrow{D} GOAL, INIT \xrightarrow{I} GOAL\}$

`r1` broadcasts S and receives:

from `r1`: `startTool(r1,S)` from `r2`: `startTool(r2,S)`

`r1` selects `startTool(r1,S)`

4. Call

$agenda = \{\langle having(r1, s), startTool(r1, S)\rangle, \langle \neg using(r1, S), startTool(r1, S)\rangle\}$
$\mathcal{CL} = \{INIT \xrightarrow{D} GOAL, INIT \xrightarrow{I} GOAL, start(r1, S) \xrightarrow{S} GOAL\}$

`r1` broadcasts `(having r1 ?t)` and receives:

from `r1`: `INIT`

`r1` selects `INIT`

5. Call

$agenda = \{\langle \neg using(r1, S), startTool(r1, S)\rangle\}$

$\mathcal{CL} = \{INIT \xrightarrow{D} GOAL, INIT \xrightarrow{I} GOAL, start(r1,S) \xrightarrow{S} GOAL, start(r1,S) \xrightarrow{S} GOAL, INIT \overset{having(r1,s)}{\rightarrow} startTool(r1,S)\}$

`r1` broadcasts `(not (using r1 S))` and receives:

from `r1`: `INIT`

`r1` selects `INIT`

6. Call

$agenda = \emptyset$

$\mathcal{CL} = \{INIT \xrightarrow{D} GOAL, INIT \xrightarrow{I} GOAL, start(r1,S) \xrightarrow{S} GOAL, start(r1,S) \xrightarrow{S} GOAL, INIT \overset{having(r1,s)}{\rightarrow} startTool(r1,S), INIT \overset{\neg using(r1,S)}{\rightarrow} startTool(r1,S)\}$

The plan after the sixth call is returned as result. The only step which needs to be executed is `startTool(r1,S)`. After the execution of this step the system is recovered as it is again in line with the stated goal. Robot `r1` is adopting the task of tightening the screws by starting the corresponding tool.

Plans to recover a system typically contain more than one step involving more than a single node which needs to execute them. In the next section a generic way is discussed how plans can be executed in a distributed environment.

Execute: distributed plan execution

When a plan is found it contains amongst other things the information $\mathcal{ST}$, the steps of the plan and $\mathcal{O}$, the ordering constraints which provide information about the coherence of a plan execution. If $s_1 < s_2 \in \mathcal{O}$, then s_1 needs to be executed before s_2. As $\mathcal{O}$ is only a partial ordering of the plan's steps it is possible that neither $s_1 < s_2$ nor $s_2 > s_1$ holds. In this case s_1 may be executed before or after s_2 - furthermore these two steps can be executed in parallel. For the parallel execution it is assumed that these steps do not interact with each other, i.e. the result of a concurrent execution is the same as executing s_1 before s_2 or s_2 before s_1.

In the developed planning engine each step of the resulting plan consists of an action and the node executing it. Figure 4.19 shows an example of such a partially ordered plan.

In this example node n_1 is supposed to execute a_1 and so on. The step (n_1, a_1) must be executed before (n_3, a_3), but the order of (n_1, a_1) and (n_2, a_2) does not matter.

If a valid plan is found the coordinator multicasts it to all nodes which are

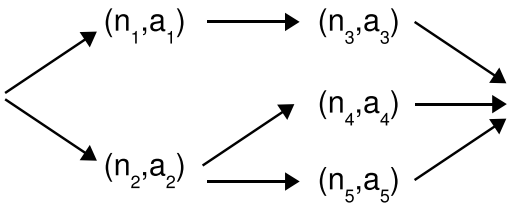


Figure 4.19: *Distributed partially ordered plan*

involved in its execution. Each node which is allowed to immediately execute a step, as it has no predecessor, perform the execution and send a message to all nodes which are scheduled to execute a subsequent step. If a node has received such a notification from all predecessors of a step, it executes this step and sends a notification itself to all successors. If a step has no successors a notification is sent to the coordinator.

In the example of Figure 4.19, the coordinator sends the plan to the nodes $n_1, \ldots, n_5$. Hereupon, node n_1 executes action a_1 and sends a message to n_3, n_2 executes a_2 and sends messages to n_4 and n_5. Upon the receipt of the notification from n_1, n_3 executes a_3 and sends a message to the coordinator. According to this, n_4 and n_5 execute their corresponding actions upon the receipt of the notification from n_2 and notify the coordinator. Having received three messages from n_3, n_4, and n_5 the coordinator is aware of the successful plan execution.

4.4.4 Extensions

After the introduction of the general functionality of the failure recovery engine, extensions are presented which aim to further improve it.

Recovery-oriented planning

The classical planning approach does not incorporate knowledge which is available due to the analyse phase. The information gathered in this phase can be used to modify the planning process to the effect that if possible only the subgoals detected as unachieved need to be considered.

Assume again the goal G has the form $g_1 \wedge \ldots \wedge g_k$, where g_i $(1 \leq i \leq k)$ is a positive or negative predicate. By the activities of the nodes during the analyse phase it is known which subgoals are unachieved and which not. As the ordering is irrelevant, suppose that $g_1 \wedge \ldots \wedge g_u$ are the unachieved goals while $g_{u+1} \wedge \ldots \wedge g_k$ are achieved. In standard planning, the POP or DPOP algorithms use the following initial parameters:

- $\mathcal{ST} = \{INIT, GOAL\}$
- $\mathcal{O} = \{INIT < GOAL\}$
- $CL = \emptyset$
- $agenda = \{g_1, \ldots, g_k\}$

Here, it is proposed to use this setting of initial parameters:

- $\mathcal{ST} = \{INIT, GOAL\}$
- $\mathcal{O} = \{INIT < GOAL\}$
- $CL = \{INIT \overset{g_{u+1}}{\rightarrow} GOAL, \ldots, INIT \overset{g_k}{\rightarrow} GOAL\}$
- $agenda = \{g_1, \ldots, g_u\}$

With this method the agenda has fewer items and the achieved subgoals are committed to be achieved by the $INIT$ step. This technique can be seen as a change of the origin of the planning process expecting it to be closer to a solution. However, it also might be possible that these initial causal links retard the planner from finding a solution and have to be removed during the planning.

A simple approach to guarantee that the planner is still complete is to first start the planning with the initial parameters set in the way proposed here. If no solution is found the planning process is reinvoked but with a standard parameter setting. A more sophisticated solution which is applied in the planning engine is to successively remove the initial causal links when the planning is coming to a dead end. This also results in the worst case to finally start with an empty set of causal links. The choice which of these initial causal links to remove is based on the "Resolve threats"-step of the planning algorithm. The causal link which caused the most threats during the addition of new causal links should be removed.

In the previous section "Plan: distributed planning" an example is stated to show the planning of the DPOP algorithm. The very same initial situation is considered here, but the recovery-oriented planning approach is applied:

1. Call

$agenda = \{\langle S, GOAL\rangle\}$
$CL = \{INIT \overset{D}{\rightarrow} GOAL, INIT \overset{I}{\rightarrow} GOAL\}$

`r1` broadcasts S and receives:

from `r1`: `startTool(r1,S)` from `r2`: `startTool(r2,S)`

`r1` selects `startTool(r1,S)`

2. Call

$agenda = \{\langle having(r1, s), startTool(r1, S)\rangle, \langle \neg using(r1, S), startTool(r1, S)\rangle\}$
$CL = \{INIT \xrightarrow{D} GOAL, INIT \xrightarrow{I} GOAL, start(r1, S) \xrightarrow{S} GOAL\}$

`r1` broadcasts `(having r1 ?t)` and receives:

from `r1`: `INIT`

`r1` selects `INIT`

3. Call

$agenda = \{\langle \neg using(r1, S), startTool(r1, S)\rangle\}$
$CL = \{INIT \xrightarrow{D} GOAL, INIT \xrightarrow{I} GOAL, start(r1, S) \xrightarrow{S} GOAL, start(r1, S) \xrightarrow{S}$
$GOAL, INIT \overset{having(r1,s)}{\rightarrow} startTool(r1, S)\}$

`r1` broadcasts `(not (using r1 S))` and receives:

from `r1`: `INIT`

`r1` selects `INIT`

4. Call

$agenda = \emptyset$
$CL = \{INIT \xrightarrow{D} GOAL, INIT \xrightarrow{I} GOAL, start(r1, S) \xrightarrow{S} GOAL, start(r1, S) \xrightarrow{S}$
$GOAL, INIT \overset{having(r1,s)}{\rightarrow} startTool(r1, S), INIT \overset{\neg using(r1,S)}{\rightarrow} startTool(r1, S)\}$

Instead of needing six calls in the best case without the application of recovery-oriented planning, in this case only four calls are needed. In more complex systems the possible reduction of the planning complexity carries much more weight.

The subgoal oriented planning is very interesting for self-healing distributed systems, as typically systems fail only partially. The technique is based on the assumption that the subgoals which still are achieved can further be achieved in the same manner. Thus, the focus of the planning is on the failures instead of considering all subgoals for the whole system.

Numerical resources

In classical planning problems numerical resources are not considered. But as the planner should be characterised by a high versatility and numerical resources are considered important in the focused domain of distributed computer systems, it has been incorporated into the planning engine. This section describes how the developed planning algorithm deals with numerical resources.

For the design of the numerical planning capabilities the trade-off between the expressivity of the planning representation and the performance of problem solving has to be considered. In this work the numerical planning capabilities are tailored to be simple enough to solve numerical planning problems with reasonable effort, but expressive enough to describe planning problems dealing with failure recovery of distributed systems. The second aspect affecting the performance of planning based failure recovery is the execution flexibility of generated plans. Particularly the concurrent execution of actions is a key for an efficient failure recovery in distributed systems. The advantages of partial order planning to generate flexibly executable plans should not be affected by the incorporation of numerical capabilities into the planning engine. PDDL, the Planning Domain Definition Language, also allows for numerical variables, basically since version 2.1. PDDL 2.1 [FL03] is separated into different levels of expressivity where level 1 is ADL planning, level 2 planning with numeric effects, level 3 planning with durative actions. However, PDDL adopts a rule called *no moving targets*, which means that two actions cannot simultaneously make use of a value if one of the two is updating the value. This is a major limitation for the plan generation. Suppose that there are two actions while one action consumes 64, the other action 32 units of the resource `Memory`. In many cases the parallel execution of both actions should be possible and beneficial but is prevented by the no moving targets rule. The planning engine of this work should be able to reason about numerical resources without applying a rule like the no moving target rule that strongly restricts the concurrency of the generated plans.

Plenty of work deals with the representation of numerical planning problems and their solution. Many modern planners are able to deal with numerical resources while the concepts to deal with them heavily depend on the basic algorithm of the particular planner. Due to space issues it is passed on a complete survey of the field of planning with numerical resources. For the interested reader it is referred to Nareyek et al. [NFF+05] as a starting point to this topic.

The ZENO planner [PW94] is a least commitment regression planner, like the planning engine of this work, that supports amongst others numerical preconditions and effects, a model of time, and continuous change. Actions are allowed to occur simultaneously only when their effects do not interfere. Due to the rather high expressivity of the representation language of ZENO its algorithm is much more complicated than the POP algorithm and incorporates methods from operations research for handling numerical constraints. The authors consider an example of a single plane able to move passengers between four possible locations to demonstrate the capabilities of ZENO. The goal is to move two passengers to one location in less than 5.5 hours. ZENO took about three minutes to solve the problem at the time

when the paper was written.

In [Koe98], Jana Koehler summarises the basic principles of reasoning about resources of the planning system IPP [KNHD97] which is based on planning graphs. In this system actions can either provide, produce, or consume resources. This rather simple perception of numerical resources that allows for more efficient planning is adopted in a similar way in this work. To deal with numerical resources the IPP planner uses resource time maps which record interval boundaries for the possible values of each resource value additionally to the planning graphs.

Numerical resources are called *fluents* in this work. The planning representation extends with the use of such fluents stated by the key word `:functions` as follows:

```
(:types TYPE_1 ... TYPE_N)

(:predicates (PREDICATE_1_NAME [ARG_1 ... ARG_N])
             (PREDICATE_2_NAME ...
     ...)

(:functions  (FUNCTION_1_NAME [ARG_1 ... ARG_N])
             (FUNCTION_2_NAME ...
     ...)

(:action ACTION_1_NAME
  [:parameters (PARAM_1 ... PARAM_N)]
  [:precondition PRECOND_FORMULA]
  [:effect EFFECT_FORMULA]
 )

(:action ACTION_2_NAME
  ...)

...


(:objects OBJ1 OBJ2 ... OBJ_N)
(:init ATOM1 ATOM2 ... ATOM_N)
(:goal PRECOND_FORMULA)
```

Fluents are stated similar to predicates, for example:

```
(:functions (ram ?n - node)
            (cpu ?n - node)
)
```

In the `:init` section additionally to Boolean predicates the values for the fluents are declared:

```
(:init
    ...
    (= (ram n1) 1024)
    (= (cpu n1) 10000)
    ...
)
```

Furthermore the `PRECOND_FORMULA` and `EFFECT_FORMULA` are also extended. A `PRECOND_FORMULA` with fluents is defined as before, but extra non negative propositions of the form `(⊙ (f) r)` connected only with $\wedge$ are allowed, where `f` is a fluent, `r` $\in \mathbb{R}$, and $\odot \in$ `{<=, <, =, >, >=}`. An `EFFECT_FORMULA` may have the additional positive propositions of the form `(⊙ (f) c)` where `f` is a fluent, `r` $\in \mathbb{R}^+$, and $\odot \in$ `{increment, decrement}`. Additionally, `r` is allowed to be a fluent whose value remains unchanged during the planning process, i.e. a constant fluent.

As an example, the following action `startService` makes use of fluents:

```
(:action startService
 :parameters   (?n - node)
 :precondition ( and (>= (ram ?n) 200))
 :effect       ( and (decrease (ram ?n) 200))
)
```

For the incorporation of techniques to enable the planner to plan with fluents it is at first assumed that the effects on fluents are *commutative*. Suppose that two actions a_1 and a_2 both have an effect that changes fluent f, then f has the same value after the execution of (a_1, a_2) as after the execution of (a_2, a_1). Furthermore, in this work it is assumed that the concurrent execution of a_1 and a_2 results in the same state as the sequential execution of either (a_1, a_2) or (a_2, a_1).

In the following, two different variants of planning with such fluents are introduced. The first variant adopts causal link planning in order to handle fluents while the second variant deviates from the methodology of causal link planning for fluents.

Causal link planning with fluents To apply causal link planning it needs to be adapted to deal with fluents. In this variant the term *resource* is considered in a narrow sense. There are three kinds of numerical actions. Actions *producing* a resource, actions *consuming* a resource, and actions *requiring* a resource. Of course, it is also possible that one action is e.g. producing one resource as well as consuming another.

Actions producing a resource increase the value of a fluent. The following action `work` is an action producing the resource `money`. The fluent `salary` is constant. It is not allowed for the fluent which is increased to appear in the precondition of that action.

```
(:action work
 :precondition ...
 :effect       (and (increase (money) (salary) ...))
)
```

Actions consuming a resource check whether in their precondition the value of a certain fluent is sufficient and consume this amount by decreasing the fluent. An instance for a consuming action is `buy` as presented below where the fluent `price` is again a constant.

```
(:action buy
 :parameters   (?i - item)
 :precondition (and (>= (money ?n) (price ?i) ...))
 :effect       (and (decrease (money) (price ?i) ...))
)
```

Different from consuming actions, requiring actions simply need the fluent to have a certain minimal value in order to be executed, but do not consume the resource. A bank could e.g. check a customer who is applying for a credit. Only if he has enough own funds he is classified creditworthy.

```
(:action applicantCheck
 :parameters   ( ?i - item)
 :precondition (and (>= (money) 10000) ...)
 :effect       (and (creditworthy))
)
```

Now it is discussed how such fluents are incorporated into the planning process.

In Step 2, the POP algorithm introduced above as Algorithm 13, selects a flaw $\langle c, a_{need} \rangle$ from the *agenda*. If c is a Boolean precondition an action

a_{add} is chosen that has c as effect, either from the steps of the plan $\mathcal{ST}$ or a new action from the set of available actions $\mathcal{A}$. If c is a numerical precondition, Step "3. Select action" needs to be adapted. The restrictions made in this section result in numerical preconditions of the only allowed form $c =$ `(>= (f) r)`. Different from Boolean preconditions where a single action/step achieves a precondition, in the numerical case it is possible that a precondition requires multiple pairs of actions/effects to achieve it. Thus, numerical causal links have the form $\{(a_{add1}, c_1), \ldots, (a_{addn}, c_n)\} \xrightarrow{c} a_{need}$, where $a_{add1}, \ldots, a_{addn}$ are actions which produce the resource f with their corresponding effects $c_1, \ldots, c_n$. Let r_i be the value f is increased by the effect c_i. First it is considered that the action a_{need} consumes the fluent f. The action/effect pairs (a_{addi}, c_i) are chosen such that $r_1 + \ldots + r_n \geq r$. Note that the action a_i can be taken from the set of steps $\mathcal{ST}$ as well as from the set of actions $\mathcal{A}$. After the application of $c_1, \ldots, c_n$ to a causal link they are updated, i.e. the corresponding values $r_1, \ldots, r_n$ by which they increase f are set to 0, except for one selected value r_i which is set to the overlapping value of $r - (r_1 + \ldots + r_n)$. If the action a_{need} does not consume but only requires the fluent f, then a dummy effect is added to a_{need}, increasing f by r.

In Step "4. Refine plan", all steps $a_{add1}, \ldots, a_{addn}$ are added to $\mathcal{ST}$, $a_{add1} < a_{need}, \ldots, a_{addn} < a_{need}$ is added to the orderings $\mathcal{O}$, and $\{(a_{add1}, c_1), \ldots, (a_{addn}, c_n)\} \xrightarrow{c} a_{need}$ to the causal links $\mathcal{CL}$. In Step "5. Update *agenda*" the numerical precondition $\langle c, a_{need} \rangle$ is removed from the agenda as in the case of a Boolean precondition. Then all preconditions of all added steps have to be added to the agenda. Step "6. Resolve threats" also needs to be modified slightly but as this is quite straightforward, the explanation is omitted here.

The introduced handling of fluents is illustrated with some examples. Consider an f producing action `p` and an f consuming action `c` as shown below. The action `p` can be used to achieve `(and (>= (f) 10))` for `c`. If `p` is selected to do so, its effect `(and (increase (f) 20))` is set to `(and (increase (f) 10))`. Thus, the resource of this step concerning `f` is half spent. Note that also the `INIT` step can be used to achieve numerical preconditions.

```
(:action p
 :precondition
 :effect       (and (increase (f) 20))
)

(:action c
 :precondition (and (>= (f) 10))
 :effect       (and (decrease (f) 10)))
```

In the next example again the precondition `(and (decrease (f) 10))` of step `p` is selected as subgoal. To achieve it, the actions `p1` and `p2` are available. Assuming that these are actions from $\mathcal{A}$ and not existing steps, multiple combinations to select the steps $\{a_{add1}, \ldots, a_{addn}\}$ are possible: $\{p1^1, p1^2\}$, $\{p1(3), p2^1, p2^2\}$, $\{p1, p2^1, p2^2(3)\}$, $\{p2^1(2), p2^2, p2^3\}$. In the case the actions produce more than needed, one action is chosen which is not decremented to 0 but set to the difference. This value represents the remaining resources of this step and is parenthesised in this example.

```
(:action p1
 :precondition
 :effect       (and (increase (f) 5))
)

(:action p2
 :precondition
 :effect       (and (increase (f) 4))
)

(:action c
 :precondition ((>= (f) 10))
 :effect       (and (decrease (f) 10))
)
```

Below is an example of a step `r` which is requiring but not consuming the resource `f`. If `p` is chosen to achieve `(>= (f) 10))` for `r`, then the effect is decremented by 10, in this case set to 0. But an effect `(increase (f) 10)` is added to `r`. This guarantees that `p` provides the resource `f` for `r` and no other step inserted before `p` is able to access the necessary amount of 10. But after the execution of `r` it is available again as an effect of `r`.

```
(:action p
 :precondition
 :effect       (and (increase (f) 10))
)

(:action r
 :precondition ((>= (f) 10))
 :effect
)
```

In the next section a different way of dealing with fluents is presented at which fluents are not protected by causal links.

Planning with unprotected fluents In this variant of planning with fluents the restrictions of the previous section do not hold. Thus possible tests in numerical preconditions are `<`, `<=`, `=`, `>=`, `>`. As a start, effects can either decrement or increment fluents.

Let $s^<$ be the set $\{s_i \in \mathcal{ST} \mid s_i < s\}$, i.e. the set of steps that are executed before step s. Analogue $s^>$ be $\{s_i \in \mathcal{ST} \mid s_i > s\}$. Furthermore let $s^\approx$ be the subset of steps $\mathcal{ST} \setminus (s^< \cup s^>)$, i.e. the steps which could be executed before or after s, or concurrently to it.

The verification whether a numerical precondition c = `(`$\odot$ `(f) r)` of step s based on a fluent `f` is currently achieved or not is calculated as follows:

1. Apply all effects of steps in $s^<$ that influence the value of `f`
2. Now it is distinguished by $\odot$, the type of the precondition `(`$\odot$ `(f) r)`.
   - $<$: Apply all effects of steps in $s^\approx$ that increment `f`. If then `f < r` holds, the precondition is certainly achieved.
   - $\leq$: Apply all effects of steps in $s^\approx$ that increment `f`. If then `f <= r` holds, the precondition is certainly achieved.
   - $=$: This case is resolved into the two preconditions: `f <= r` and `f >= r`.
   - $\geq$: Apply all effects of steps in $s^\approx$ that decrement `f`. If then `f >= r` holds, the precondition is certainly achieved.
   - $>$: Apply all effects of steps in $s^\approx$ that decrement `f`. If then `f > r` holds, the precondition is certainly achieved.

As an example to clarify this verification process the following two actions are considered:

```
(:action p
 :precondition (and (<= (f) 0))
 :effect       (and (increase (f) 1))
)

(:action c
 :precondition (and (> (f) 0))
 :effect       (and (decrease (f) 1))
)
```

Suppose that `f` is 0 in the initial state. The steps of the current plan are $\mathcal{ST} = \{c^1, c^2, p^1, p^2\}$. The only ordering constraint in $\mathcal{O}$ is $p^1 < c^1$. Figure 4.20 illustrates this plan.

To check now whether the precondition $f > 0$ of step c^1 holds, first all effects of steps in $c^{1<}$, i.e. steps that are executed before c^1 and influence `f`

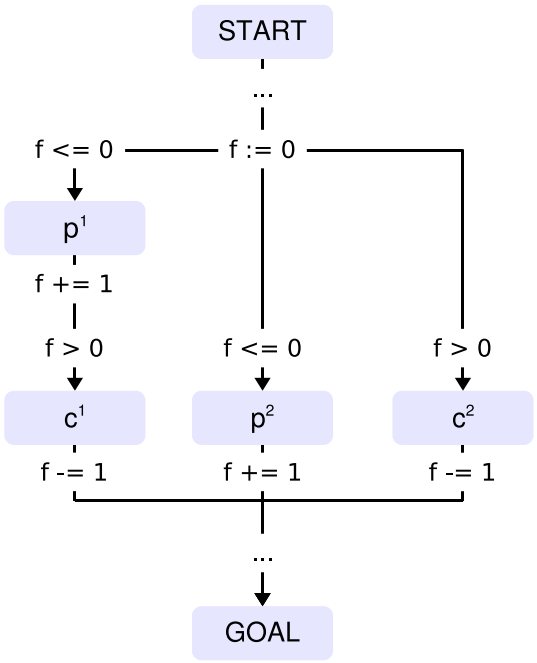


Figure 4.20: *Planning with numerical resources*

are applied. After this, `f` has a value of 1, as p^1 is executed before c^1. Now, all effects of steps in $c^{1\approx}$ that decrement `f` are applied; this is the effect of c^2 in this example that decrements `f` by one, and this results in a value of 0. Thus $f > 0$ is not achieved as c^2 could be executed before c^1 and destroy the precondition. There are three ways to achieve the precondition for the current plan:

1. Add a new step p^3 to the plan together with the ordering constraint $p^3 < c^1$.
2. Add the ordering constraint $p^2 < c^1$ to $\mathcal{O}$.
3. Add the ordering constraint $c^1 < c^2$ to $\mathcal{O}$.

It is not necessary that the mathematical expression `r` is constant, but it is not allowed that it is updated by any action in $s^{\approx}$. This means its current value is known.

The most important changes of the POP algorithm, using the way of planning with fluents as presented here, are:

1. **Termination:** If *agenda* is empty and all numerical preconditions are achieved, return $\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL} \rangle$. Note that the set of causal links $\mathcal{CL}$ and the *agenda* is only used for non-numerical constructs.
2. **Select subgoal:** Choose either an item $\langle c, a_{need} \rangle$ from *agenda* or an unachieved numerical precondition as subgoal.

3. **Select action:** If a non-numerical subgoal is selected, the steps 3., 4., and 5. stay the same as in the original POP algorithm. Therefore, only the numerical case is presented: Select an action either from the steps of the plan $\mathcal{ST}$ or a new action from the set of available actions $\mathcal{A}$ that *helps* to fulfil the precondition. If no such actions exists, return a failure. An action helps to fulfil a precondition $f \odot \mathtt{r}$ if it has an effect e as follows:
   - $\odot$ **is either '$<$' or '$\leq$':** The effect e decrements `f` or increments `r`.
   - $\odot$ **is '$=$':** The effect e reduces the value of $|f - \mathtt{r}|$.
   - $\odot$ **is either '$>$' or '$\geq$':** The effect e increments `f` or decrements `r`.
4. **Refine plan:** Add a_{add} and an ordering constraint. $\langle \mathcal{ST}, \mathcal{O}, \mathcal{CL} \rangle := \langle \mathcal{ST} \cup \{a_{add}\}, \mathcal{O} \cup \{a_{add} < a_{need}\}, \mathcal{CL} \rangle$.

If the commutativity of the fluents does not hold, the determination whether a precondition is achieved or not is much harder. The verification of a numerical precondition in a commutative environment utilises this feature to compute the "worst case" ordering, and this enables a fast computation. Unfortunately, without the commutativity assumption this worst case cannot be computed easily. Therefore, there is nothing else for it but to have a look on every possible combination of actions that can be executed before the action of the precondition that is checked.

Suppose the following two actions where c is slightly modified and additionally to the decrease and increase, a setting of fluents to a specific value is allowed.

```
(:action p
 :precondition (and (= (f) -1))
 :effect       (and (increase (f) 1))
)

(:action c
 :precondition (and (> (f) 0))
 :effect       (and (decrease (f) 1))
)
```

The action `p` does not decrement the fluent value but sets it to -1. This violates the commutativity of the actions. If e.g. `f` has a value of 0 and c is executed before p the resulting value is again 0. The other way round, if p is executed before c then `f` is -1 after the execution.

To verify a precondition p with fluent `f` of step s in the non-commutative case, the following set of steps is considered: $s^* := s^{\approx} \cup s^{<} \cup \{s\}$, where all steps are removed from $s^{\approx}$ and $s^{<}$ that have no influence on `f`. This are all

steps that could influence the precondition c. As this influence depends on the order of the actions, all possible orders, i.e. all linearisations regarding $\mathcal{O}$, are computed. The precondition c is achieved if it is achieved for all linearisations. The set of all linearisations can be computed using e.g. the algorithm of Varol and Rotem [VR81].

Suppose that there are three actions a, b, c with no ordering constraints. Then the set of linearisations is $\{(a,b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a)\}$. An ordering constraint $a < b$ reduces the set of linearisations to $\{(a,b,c), (a,c,b), (c,a,b)\}$, the ordering constraints $a < b$, $a < c$, and $b < c$ to $\{(a,b,c)\}$.

Consider an example plan analogue to the example in Figure 4.20, with the four steps $\mathcal{ST} = \{c^1, c^2, p^1, p^2\}$ and the ordering constraint $\mathcal{O} = \{p^1 < c^1\}$. The difference to the previous example is that c is defined in a way that violates the commutativity. To check whether the precondition $f > 0$ of step c^1 holds, first c^{1*} is composed: $\{p^2, c^2\} \cup \{p^1\} \cup \{c^1\}$. The set of all possible linearisations is as follows:

$\{(p^1,p^2,c^1,c^2), (p^1,p^2,c^2,c^1), (p^1,c^1,p^2,c^2), (p^1,c^1,c^2,p^2), (p^1,c^2,p^2,c^1), (p^1,c^2,c^1,p^2), (p^2,p^1,c^1,c^2), (p^2,p^1,c^2,c^1), (p^2,c^2,p^1,c^1), (c^2,p^1,p^2,c^1), (c^2,p^1,c^1,p^2), (c^2,p^2,p^1,c^1)\}$

If now all 4-tuples of the set of linearisations are cut before c^1, the resulting set represents all possible orderings of actions that can occur before c^1 and influence the precondition:

$\{(p^1,p^2), (p^1,p^2,c^2), (p^1), (p^1,c^2,p^2), (p^1,c^2), (p^2,p^1), (p^2,p^1,c^2), (p^2,c^2,p^1), (c^2,p^1,p^2), (c^2,p^1), (c^2,p^2,p^1)\}$

Now it is checked if the precondition is fulfilled for every possible execution of steps before c^1. This validation can be seen in Table 4.1(a). The precondition is not achieved as it is not fulfilled for every linearisation.

One possible refinement to fulfil the precondition $f > 0$ is to add the ordering constraint $c^1 < c^2$ to $\mathcal{O}$. The resulting validation is presented in Table 4.1(b).

The test whether a numerical precondition is achieved in an environment without commutativity of actions is much more costly.

Action costs

It can be beneficial to assign costs to actions as this provides a method to influence planning decisions. If for instance two actions have similar effects and preconditions but one of these should be preferred, a lower action cost could be assigned to it. The planning process is responsible to analyse such

(a) Precondition not achieved

(p^1, p^2)	2	✔
(p^1, p^2, c^2)	-1	✘
(p^1)	1	✔
(p^1, c^2, p^2)	0	✘
(p^1, c^2)	-1	✘
(p^2, p^1)	2	✔
(p^2, p^1, c^2)	-1	✘
(p^2, c^2, p^1)	0	✘
(c^2, p^1, p^2)	1	✔
(c^2, p^1)	0	✘
(c^2, p^2, p^1)	1	✔

(b) Precondition achieved

(p^1, p^2)	2	✔
(p^1)	1	✔
(p^2, p^1)	2	✔

Table 4.1: *Verification of a numerical precondition in a non-commutative environment*

action costs. In the following section it will become clear how the failure recovery engine handles this.

With the consideration of costs, the definition of actions changes slightly:

```
(:action ACTION_NAME [:costs COSTS]
   [:parameters (PARAM_1 ... PARAM_N)]
   [:precondition PRECOND_FORMULA]
   [:effect EFFECT_FORMULA]
)
```

Flaw and plan selection

Partial order planning has many advantages like a high flexibility and support of expressive planning definition languages. The main reason to decide in favour of a partial order causal link planning scheme is the generation of partially ordered plans which can be executed efficiently and flexible, particularly in distributed environments. State-space planners do not have the ability to find parallel plans in an efficient way [Has00]. CSP planners like GRAPHPLAN only output a very restricted class of parallel plans [NK01].

However, the plain POP algorithm has a poor performance in comparison to the fastest heuristic state-space planners like UNPOP and CSP planners like GRAPHPLAN respectively. To overcome this issue, sophisticated flaw and plan selection strategies can be applied. The flaws of a plan are the union of its open conditions and unsafe causal links, also called threats.

Flaw and plan selection help the recovery engine to efficiently search

through the space of plans. The POP algorithm presented before in Algorithm 13 does not clarify the search character of the planning process but represents one single planning step. At several points of the algorithm, decisions are taken like the selection of a certain open condition/subgoal and the selection of an action to achieve it. However, to ensure the completeness of the algorithm, if the planner comes to a dead end, the planner must backtrack to try other options.

In Algorithm 15, a different notation of the POP algorithm is presented which does not focus on a single planning step but on the search strategy. It is stated similarly to [PJP97].

Algorithm 15 Search scheme of the POP algorithm

```
1:  function POP
2:      N = {NULL_PLAN}
3:      while N ≠ ∅ do
4:          π = choose and remove plan from N          ▷ Plan selection
5:          if π is solution then return π             ▷ Solution found
6:          else
7:              select flaw φ from π                   ▷ Flaw selection
8:              add all possible refinements of π,
9:              in order to achieve φ, to N
10:         end if
11:     end while
12:     return 'failure'                               ▷ Problem lacks solution
13: end function
```

The flaw selection of Line 7 basically corresponds to "Select subgoal" and "6. Resolve threats" of Algorithm 13. The plan selection of Line 4 is covered by "3. Select action" in Algorithm 13.

The search decisions - plan and flaw selection - have a great influence on the planning performance. The plan selection strategy often is implemented as a search heuristic of the A^* [HNR72] search. Therefore, the term planning *heuristic* is used synonymously for *plan selection*.

Poet et al. [PS93] demonstrate that the planning efficiency of partially ordered planners may be vastly improved by the use of alternative threat removal strategies. Common threat removal strategies are to remove threats as they are discovered, used e.g. in SNLP [MR91], or to delay the threat resolution partially or completely like in SIPE [Wil88].

Joslin et al. [JP94] describe a least-cost flaw repair strategy to guide the flaw selection during planning. This can be seen as a generalisation of [PS93] in which the least-cost flaw repair strategy treats all flaws uniformly.

Gerevini et al. [GS96] propose techniques to accelerate partial order plan-

ners. Therefore they propose to adjust the default A^* plan selection heuristic used by UCPOP, to apply *zero commitment* plan refinements and to prune the search.

REPOP [NK01] of Nguyen and Kambhampati is a partial-order planner based on UCPOP. It is based on the key insight that techniques responsible for the efficiency of other planning paradigms like distance based heuristics, reachability analysis, and disjunctive constraint handling can also be adapted to dramatically improve the POP algorithm. Their results show that REPOP outperforms GRAPHPLAN in several parallel domains. Moreover the plans tend to be more flexible than those generated by state-space planners or GRAPHPLAN.

VHPOP [YS03b] is a partial order causal link planner which combines known and proposes new flaw and plan selection techniques resulting in a planner competitive with established CSP-based and heuristic state-space planners.

Especially REPOP and VHPOP show that the performance of partial order planners can keep up with planners that are based on other paradigms. The focus in developing the planning engine has not been on performance improvements using well investigated standard techniques. The intention of this work is to propose a framework for self-healing. To incorporate complex flaw and plan selection techniques as used in VHPOP is not scope of this work and might be addressed in the future.

The following flaw and plan selection strategies are currently available for the planning engine.

Flaw selection A flaw is either an open condition or a threat. Whenever a threat arises in a partial plan, it is addressed before remaining open conditions. For open conditions and the case of more than one threat, two simple flaw selection strategies are implemented:

FIFO: Flaws arising first are selected first.

LIFO: Flaws arising last are selected first.

Plan selection The planning engine uses an A^* search. This algorithm requires a heuristic function $f(\pi) = g(\pi) + h(\pi)$. The function $g(\pi)$ represents the cost of getting from the initial plan to the investigated plan π. $h(\pi)$ is an estimate for the remaining costs to reach the goal, a complete plan.

The partial order causal link plan π consists of a set of steps $\mathcal{ST}$, ordering constraints $\mathcal{O}$, and causal links $\mathcal{CL}$. The set $\mathcal{OC}$ is the set of open conditions of the plan and corresponds to the list *agenda* in Algorithm 13. The

following plan selection strategies are available:

Simple

$g(\pi)$: Sum of the action costs of steps within $\mathcal{ST}$.

$h(\pi)$: Number of open conditions, i.e. $|\mathcal{OC}|$.

Achieve

$g(\pi)$: Sum of the action costs of steps within $\mathcal{ST}$.

$h(\pi)$: Only those open conditions in $|\mathcal{OC}|$ which do not match to any effect of any step within $\mathcal{ST}$ are counted. Thus it is taken into account that some open conditions might be achieved by just linking an existing step to it.

EqualDist Simple domain independent heuristic designed for a distributed planning process. It aims to ensure the equal distribution of plan steps among the nodes of the system.

$g(\pi)$: Sum of the action costs of steps within $\mathcal{ST}$, and multiple step assignments to nodes are penalised.

$h(\pi)$: Number of open conditions in $|\mathcal{OC}|$

The same definition of the estimation function $h(\pi)$ as used in *Simple* and *EqualDist* is proposed in [SG95]. The definition of the cost function $g(\pi)$ in *Simple* and *Achieve* is similarly used in VHPOP [YS03b], here with additional consideration of action costs.

Note that the introduced flaw and plan selection techniques are quite simple. Deriving more sophisticated techniques especially adapted to distributed planning seems to be an interesting field of research and might be addressed in future work. The main purpose of the introduced strategies for flaw and plan selection is to provide a better classification of the evaluation results, presented in the following section.

4.5 Evaluation

In this section an evaluation for the introduced failure recovery engine is provided. It serves as a proof of concept for the proposed mechanisms and demonstrates benefits of the presented extensions.

The evaluation is conducted on the basis of two scenarios. These are derived from applications of two projects which are part of the priority programme "Organic Computing" [ACE+03] funded by the German Research Foundation (DFG). The chosen application scenarios are used within these projects to demonstrate developed Organic Computing principles.

The first scenario, a simple production cell, is taken from the project "Formal Modeling, Safety Analysis, and Verification of Organic Computing Applications (SAVE ORCA)", led by Prof. Dr. Wolfgang Reif from the University of Augsburg. The project deals with the systematic, top-down design and construction of highly reliable and adaptive OC applications.

The second scenario, a smart office/building environment called "Smart Doorplates", is taken from the project "OCμ - Organic Computing Middleware for Ubiquitous Environments", led by Prof. Dr. Theo Ungerer from the University of Augsburg. The main goals of this project are the design of a middleware toolkit and the investigation of self-x techniques.

The evaluation presented in the following has been conducted in the course of Thorge Andersen's diploma thesis [And08].

4.5.1 Production cell scenario

The investigated production cell scenario represents a modified version of the production cell used before in Section 4.4.3 to explain the planning engine's functionality. It is more focused on the processing of single workpieces and the specification of the carts' `transport` action is directly based on the robots itself rather than on their tasks. This way to model the production cell is more closely related to the currently used production cell scenario of SAVE ORCA.

The goal of the production cell, consisting of three robots (`r1`, `r2`, `r3`), and two carts (`c1`, `c2`), is to correctly process workpieces. Therefore, a hole needs to be drilled, a screw inserted, and tightened whereas any robot has three corresponding tools. The predicates `D`, `I`, `T` describe whether a workpiece has already been processed by the drilling, the inserting, or the tightening tool. The predicate `wpat` indicates the location of a workpiece, `task` declares whether a robot already is using a tool, while `transport` states similarly for carts whether they already adopt a transport task.

The goal of the production cell is to produce workpieces having a tightened screw:

```
(:goal (T))
```

It is assumed that initially no tasks are assigned. Any node, either a robot or a cart, has only local information. The view of e.g. robot `r1` is as follows:

```
(:types robot cart - object)
(:predicates (wpat ?r - robot) (task ?r - robot) (D) (I) (T))

(:init
 (not (D))
 (not (I))
 (not (T))
 (wpat r1)
 (not (task r1))
)

(:action drill
  :precondition (and (not (D)) (wpat r1) (not (task r1)))
  :effect       (and (D) (task r1))
)
(:action insert
  :precondition (and (not (I)) (D) (wpat r1) (not (task r1)))
  :effect       (and (I) (task r1))
)
(:action tighten
  :precondition (and (not (T)) (I) (wpat r1) (not (task r1)))
  :effect       (and (T) (task r1))
)
```

The view of a cart is different, `c1` for instance has the following environmental information:

```
(:types robot cart - object)
(:predicates (wpat ?r - robot) (task ?r - robot)
             (D) (I) (T) (transporting ?c - cart)
)

(:init
 (not (transporting c1))
)

(:action transport
  :parameters   (?r1 ?r2 - robot)
  :precondition (and (wpat ?r1) (not (wpat ?r2))
                (not (transporting c1)))
  :effect       (and (not (wpat ?r1)) (wpat ?r2)
                (transporting c1))
)
```

The different views of the nodes result from each node having a local partial view on the system.

The resulting partial order plan of a distributed planning process, which works as described in Section 4.4.3, applied to the production cell looks as follows:

(`r1`, `drill`) → (`c1`, `transport r1 r2`) → (`r2`, `insert`) →
(`c2`, `transport r2 r3`) → (`r3`, `tighten`)

For evaluation purposes the planning engine configures the initial production cell using different flaw and plan selection techniques. Figure 4.21 illustrates the results of these experiments.

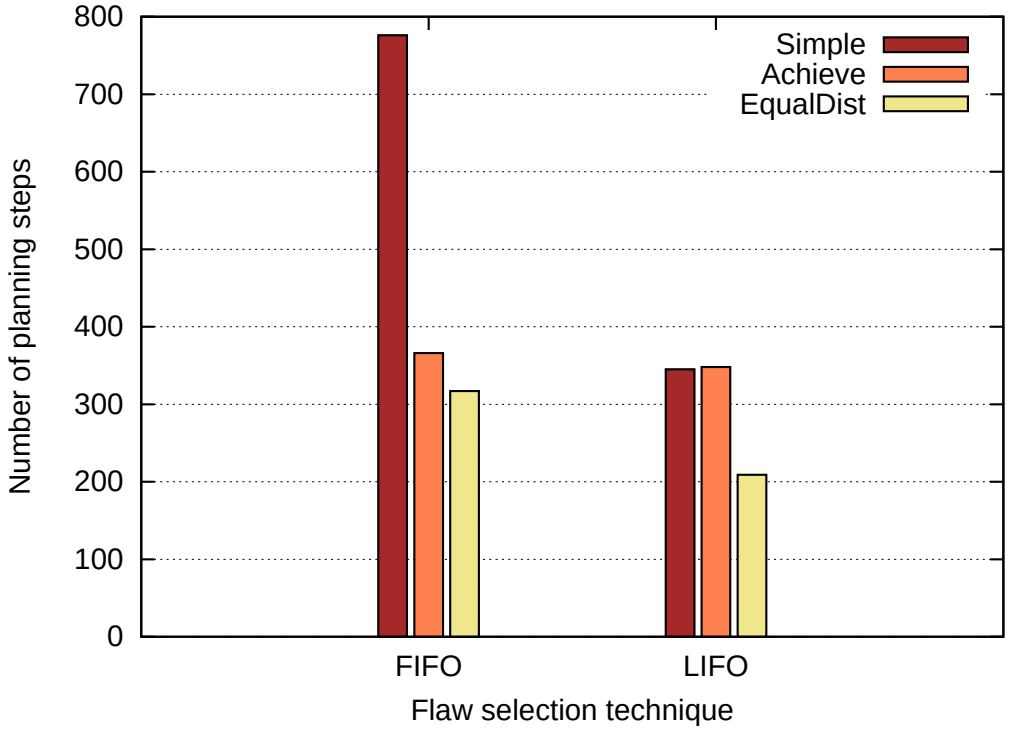


Figure 4.21: *Evaluation results: Production cell*

The y-axis shows the number of planning steps, more precisely the number of partial plans which are refined until a valid plan is found. Fewer steps represent a more efficient planning process. Different flaw (x-axis) and plan selection (see legend) strategies as proposed in the previous section are applied.

The *LIFO* outperforms the *FIFO* flaw selection in all experiments. The *EqualDist* plan selection proposed in this work as a simple heuristic for distributed planning achieves the best results compared to *Simple* and *Achieve*.

In the following the much more complex and dynamic Smart Doorplate scenario is discussed where the self-healing and failure recovery aspects will be given more prominence.

4.5.2 Smart Doorplate scenario

The Smart Doorplate [TBPU03b] project envisions the use of smart doorplates within an office building. The doorplates are amongst others able to display current situational information about the office owner and to direct visitors to his current location based on a location-tracking system.

The Smart Doorplate system in the variant discussed here consists of *sensor nodes* to provide basic location-tracking functionality and *service nodes* which are able to host different kinds of services. The system is arranged into *sections* and one section normally contains three sensor nodes. To ensure adequate location-tracking, per section at least one active sensor node is needed. The bigger the Smart Doorplate system the more service nodes are necessary. A system with n sections typically has n service nodes. The computers within the offices could for example be used as such.

Two basic location-tracking techniques based on radio or supersonic exist, while the latter usually provides better results. Sensor nodes have the technical equipment for either of them or both.

There are three different types of services supposed to run on service nodes. *Location-tracking services* compute locations of humans in the smart environment based on the raw data sensor nodes are providing. *Prediction services* are able to predict the next location of humans based on their habits. A *hybrid prediction service* uses several different prediction services as input to provide a higher quality prediction.

The following predicates are used to describe sensor nodes:

hasRadio: Sensor node is equipped with radio unit.

hasSupersonic: Sensor node is equipped with supersonic unit.

radio: Sensor node's radio unit is active.

supersonic: Sensor node's supersonic unit is active.

providesLoc: Sensor node provides location-tracking data (either with radio or with supersonic unit).

inSection: Assignment of sensor nodes to sections.

Numerical resources are needed to model service nodes:

LTS: Number of location tracking services.

PS: Number of prediction services.

HPS: Number of hybrid prediction services.

SEC: Constant number of sections of the system.

ram: Currently available memory of a service node.

cpu: Currently available CPU resources of a service node.

numLTS: Number of location tracking services on a service node.

numPS: Number of prediction services on a service node.

numHPS: Number of hybrid prediction services on a service node.

The system's goal is stated as follows:

```
(:goal (and (forall (?sec - section)
              (exists (?n - sensornode)
                (providesLoc ?n ?sec)))
            (>= (HPS) 1)
            (>= (LTS) (SEC)))
)
```

This means, (1) in all sections at least one sensor node must provide a location-tracking functionality. (2) In the whole system at least one hybrid predictor must be available. (3) The number of location-tracking services which are processing the data of active sensor nodes must be greater or equal than the number of sections. The sensor nodes are responsible to check property (1) for their corresponding section, the service nodes monitor (2) and (3).

The information available to a sensor node is demonstrated by the example of `sensor_n1` for a Smart Doorplate system consisting of two sections:

```
(:types
  node section - object
  sensornode servicenode - node
)
(:predicates
 (hasRadio ?sn - sensornode)
 (hasSupersonic ?sn - sensornode)
 (radio ?n - sensornode)
 (supersonic ?n - sensornode)
 (providesLoc ?n - sensornode ?sec - section)
 (inSection ?n - sensornode ?sec - section))
```

```
(:objects
  sec1 sec2 - section
  sensor_n1 sensor_n2 - sensornode
)

(:init
 (hasRadio sensor_n1)
 (not (radio sensor_n1))
 (not (hasSupersonic sensor_n1))
 (not (supersonic sensor_n1))
 (not (providesLoc sensor_n1 sec1))
 (not (providesLoc sensor_n1 sec2))
 (inSection sensor_n1 sec1)
 (not (inSection sensor_n1 sec2))
)

(:action startLocRadio : 1.1
  :parameters (?sec - section)
  :precondition (and (not (providesLoc sensor_n1 ?sec))
                     (hasRadio sensor_n1)
                     (inSection sensor_n1 ?sec))
  :effect       (and (providesLoc sensor_n1 ?sec)
                     (radio sensor_n1))
)
(:action startLocSupersonic
  :parameters   (?sec - section)
  :precondition (and (not (providesLoc sensor_n1 ?sec))
                     (hasSupersonic sensor_n1)
                     (inSection sensor_n1 ?sec))
  :effect       (and (providesLoc sensor_n1 ?sec)
                     (supersonic sensor_n1))
)
(:action stopLocRadio : 0.9
  :parameters   (?sec - section)
  :precondition (and (radio sensor_n1))
  :effect       (and (not (providesLoc sensor_n1 ?sec))
                     (not (radio sensor_n1)))
)
(:action stopLocSupersonic
  :parameters   (?sec - section)
  :precondition (and (supersonic sensor_n1))
  :effect       (and (not (providesLoc sensor_n1 ?sec))
                     (not (supersonic sensor_n1)))
)
```

Thus, sensor nodes have a quite restricted view, basically concerning only their own status. Their main purpose is to provide location-tracking functionality within a section by starting either the radio or the supersonic unit. To prefer the supersonic unit, the starting action of the radio unit has a slightly higher execution cost. The contrary is holding for the corresponding stopping actions.

The information available to a service node is demonstrated by the example of the node `service_n1`:

```
(:types
  node section - object
  sensornode servicenode - node
)
(:functions
  (ram ?n  servicenode)
  (cpu ?n  servicenode)
  (numLTS ?n  servicenode)
  (numPS ?n  servicenode)
  (numHPS ?n  servicenode)
  (LTS) (PS) (HPS) (SEC)
)
(:objects
  sec1 sec2 - section
)

(:init
 (= (ram service_n1) 512)
 (= (cpu service_n1) 500)
 (= (LTS) 1)
 (= (PS) 3)
 (= (HPS) 1)
 (= (SEC service_n1) 2)
 (= (numLTS service_n1) 1)
 (= (numPS service_n1) 1)
 (= (numHPS service_n1) 0)
)

(:action startLTS
  :precondition (and (>= (ram service_n1) 150)
                     (>= (cpu service_n1) 200))
  :effect       (and (decrease (ram service_n1) 150)
                     (decrease (cpu service_n1) 200)
                     (increase (numLTS service_n1) 1)
                     (increase (LTS) 1))
)
```

```
(:action startPS
  :precondition (and (>= (ram service_n1) 200)
                     (>= ( cpu service_n1) 200))
  :effect       (and (decrease (ram service_n1) 200)
                     (decrease (cpu service_n1) 200)
                     (increase (numPS service_n1) 1)
                     (increase (numPS) 1))
)
(:action startHPS
  :precondition (and (>= (ram service_n1) 512)
                     (>= ( cpu service_n1) 350)
                     (>=(PS) 3))
  :effect       (and (decrease (ram service_n1) 512)
                     (decrease (cpu service_n1) 350)
                     (increase (numHPS service_n1) 1)
                     (increase (HPS) 1))
)
```

Service nodes are able to start and stop location-tracking, prediction, and hybrid prediction services. The stopping actions are omitted for a shorter presentation. To start any service, enough resources have to be available. As further requirement, for hybrid prediction services it is declared that at least three prediction services must be running somewhere in the system.

Consider a system consisting of two sections, with three sensor nodes per section and a total of two service nodes. Assume furthermore that the system is in a "blank" state - all sensor nodes are inactive and no service is running on the service nodes. The resulting plan for such a situation might look as illustrated in Figure 4.22.

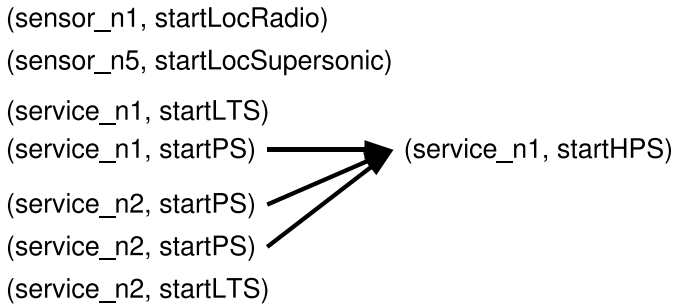


Figure 4.22: *Plan to configure the Smart Doorplate system*

Albeit the recovery engine is capable of configuring the system in its initial state, the focus of this work is to enable self-healing. The evaluation conducted using the Smart Doorplate scenario takes this into account. To investigate the properties of the failure recovery engine, properly running

Smart Doorplate systems of different sizes are considered, while a failure injection component simulates node failures.

The investigated scenarios are denominated as follows:

#Sections - #Node failures

In detail, Scenario $x - y$ means that a operating Smart Doorplate system consisting of x sections is considered and the failure injection simulates the simultaneous failure of y random nodes. Cases impossible to recover the system are omitted, like for instance a complete failure of all service nodes. The failure recovery engine recovers the system and it is measured how many planning steps are necessary to find a valid recovery plan. To ensure representative values, any scenario is repeated 100 times and the results are averaged. The experiments range from a system with two sections and a single node failure $(2 - 1)$ to a system with 32 sections and 8 node failures $(32 - 8)$. As any section entails four nodes, three sensor and one service node, the investigated systems range from 8 to 128 nodes. Figure 4.23 presents the evaluation results.

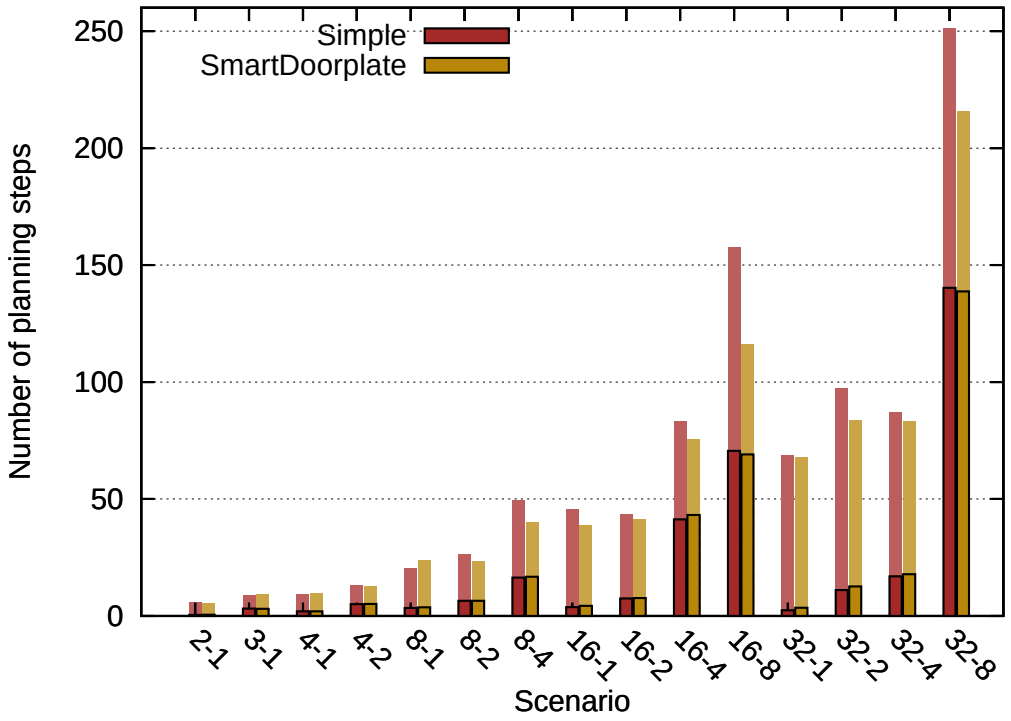


Figure 4.23: *Evaluation results: Smart Doorplate*

The y-axis shows the average number of planning steps. The x-axis describes the different scenarios in the syntax mentioned above. As flaw selection technique only *LIFO* is illustrated which performs better than *FIFO* in all cases. The legend contains the two applied plan selection strategies,

Simple and *Smart Doorplate*. The former is the above introduced domain independent heuristic, while the latter is a new domain dependent heuristic which performed best compared to *Simple*,*Achieve*, and *EqualDist*. It is adapted to the Smart Doorplate scenario and it is based on *EqualDist* but additionally penalises service stopping actions (`stopPS`, `stopHPS`, `stopLTS`). In Figure 4.23, the columns in the background without borders correspond to the planning engine without recovery-oriented planning, the bordered columns in the foreground represent the planning engine using recovery-oriented planning as proposed in Section 4.4.4.

Recovery-oriented planning which is based on the incorporation of knowledge accumulated in the analyse phase influences the results much more than any flaw or node selection strategy. Especially if only small parts of the system fail the savings applying recovery-oriented planning are enormous. In the scenario $32 - 1$ and the *Simple* heuristic the recovery-oriented planning saves about 97% of the planning steps.

The introduced extension of planning with numerical resources provides the possibility to apply the self-healing approach to the Smart Doorplate application. Note that the variant "Causal link planning with fluents" is used in the evaluation to plan with numerical resources as it allows a better cooperation with recovery-oriented planning.

In the Smart Doorplate application, the possibility to assign costs to actions is used to influence the recovery engine to prefer supersonic to radio sensor units.

The planning for the scenario $32 - 8$ without recovery-oriented planning and the *Simple* heuristic, i.e. the most complex setting, took on average 16.5 seconds and in the worst case 81.5 seconds. The simulations have been conducted on a standard PC with an Intel® Core™ 2 Quad processor with 2.4 GHz, whereas the simulation environment is not designed to use more than one core.

The modelling of the Smart Doorplate domain as presented here mainly deals with the consistent assignment of services to nodes but does not consider data recovery after a node failure with subsequent reconfiguration. Especially the prediction services maintain a knowledge base consisting of movement patterns of persons. If such information is stored only locally on the current host of a service, a node failure results in the loss of that data. Restored services have to start from scratch when recovered. To avoid that, the Smart Doorplate environment provides a distributed data store [TEP+07] which allows a safe and redundant storage of data distributed over the network. Alternatively to the usage of this data store, all nodes monitoring another node (see Chapter 3) could be made responsible to mirror the knowledge bases of services running on the latter. Every time a service stores a new movement pattern into its own knowledge base, an automatic notifi-

cation of the monitoring nodes is triggered, containing this new entry to update the remote instances.

4.6 Conclusions and future work

In this chapter a failure recovery engine based on automated planning is proposed. It is a goal-oriented approach where a description is provided for the system which defines its desired properties. It is left to the recovery engine to monitor the system, and, if necessary, generate a plan to recover the system and perform the plan's steps. As the approach is based on a sound and complete algorithm it can be trusted in the behaviour of the system: If a solution exists it is guaranteed that the system finds it, and all successful executions of generated plans lead to a desired state.

A technique is proposed which allows a distributed consistency check of system objectives specified in PDDL. A distributed planning algorithm called DPOP has been developed to enable the nodes of a distributed system to derive a plan which recovers the system in the case of a violation of the system's objectives. The presented plan execution method provides the ability to concurrently execute plans in distributed systems.

Several extensions are presented which aim to provide extra functionality or better self-healing performance. Recovery-oriented planning incorporates knowledge accumulated during the monitoring phase to improve the planning process. Two new approaches to plan with numerical resources are proposed, and one of these approaches uses partial order causal link planning for numerical resources. This allows a consistent treatment of numerical and non-numerical facts for partial order planners. The developed planning engine supports action costs as well as the usage of flaw and plan selection strategies.

The functionality of the proposed approach has been evaluated within different scenarios. The production cell scenario shows that the failure recovery approach is also appropriate for configuring a system from its starting point. It clarifies how different plan and flaw selection can be used and how they influence the planning performance. The Smart Doorplate scenario gives a proof of concept that the approach is suitable to autonomically manage quite complex systems with more than 100 nodes. The recovery-oriented planning approach proposed in this work saved up to 97% of overhead in the planning process, much more than any flaw and node selection technique. The modelling of the Smart Doorplate scenario uses the numerical planning features as well as the possibility to assign action costs.

The dissimilarity of the two investigated evaluation scenarios indicate the

broad capabilities of the proposed failure recovery approach. The limits of the introduced approach are defined only by the expressivity of the subset of PDDL used to model domains. Furthermore, the planning complexity has to be considered which depends on the complexity of a domain and the way it is modelled.

Interesting starting points for future work are more sophisticated flaw and node selection strategies for distributed systems which might dynamically adapt to the actual environment. The next chapter provides a further discussion of future work.

5

Towards an architecture for highly complex systems

5.1 Introduction

In this chapter, ideas for a generic architecture of self-healing distributed systems are presented. It can be seen as an outlook for this dissertation and extends the concepts of the previous chapters. The aim is to provide self-healing capabilities for highly complex distributed systems. It is proposed that each entity of a distributed system is managed by a so called *node manager*. For the design of these components, psychological concepts have been consulted. Sociological ideas form the basis of the cooperation capabilities of these distributed managers.

5.2 Survey of psychological and sociological concepts

As psychological and sociological concepts are consulted, these are introduced in the following mainly taken from Zimbardo et al. [ZG99].

5.2.1 Psychological concepts

The human *nervous system* refers to all nerve cells in the body. The nervous system consists of two major parts: the *central nervous system* (CNS) and the *peripheral nervous system* (PNS). The CNS represents the largest part of the nervous system, and consists of the *brain* and the *spinal cord*. The CNS is like a central control tower which controls the permanent "approach" of stimuli and "departure" of commands and reactions [ZG99]. The spinal cord connects the brain with the PNS and has connections to sensory receptors of the whole body, to muscles, and to glands. The spinal cord is also responsible for simple reflexes without involvement of the brain: Animals whose brain is disconnected from its spinal cord are able to pull back a leg from a painful stimulus [ZG99]. The brain is normally informed about such events but the action is executed without it.

The PNS is the part of the nervous system that resides outside the CNS. It is a network of sensory and motor neurons and transmits information to and from the CNS.

Figure 5.1 summarises the hierarchical structure of the nervous system.

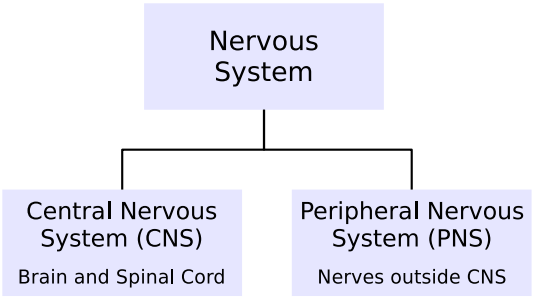


Figure 5.1: *The human nervous system*

The nervous system is made up of electrically excitable cells, the neurons. They are the basic building blocks of the nervous system and can be divided into three functional classes:

Sensory neuron: Also called afferent neuron. Inward neuron that sends information from the sensory organs, through the nerves, into the CNS.

Motor neuron: Also called efferent neuron. Outward neuron that transmits messages from the CNS to the muscles and glands.

Interneurons: These neurons are located in the CNS and connect other neurons.

The simplest form of behaviour controlled by the nervous system are automatic reflexes which can occur without an involvement of the brain. An example is the reflectory pull back of a finger after a painful stimulus due

to touching something sharp or hot. If the receptors for pain are stimulated they send information via sensory neurons to an interneuron which is located in the spinal cord. The interneuron itself stimulates motor neurons activating muscles that cause the pulling back of the finger from the pain-causing object. After the reaction the brain is informed. Thus, the body is able to react very quickly to prevent injuries due to the reflective process and the brain is able to store this experience afterwards to avoid such a situation in the future. The *reflex arc*, the processing pathway for a reflex is illustrated in Figure 5.2.

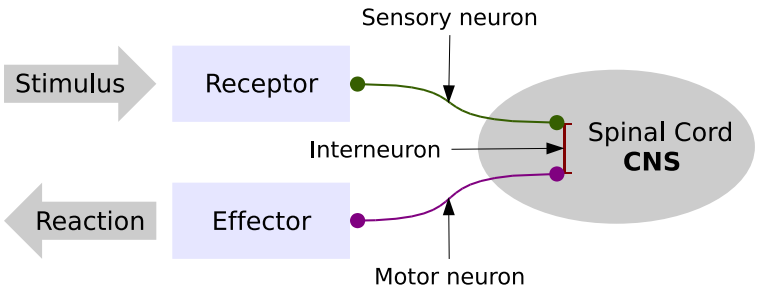


Figure 5.2: *Simple reflex arc*

A reflex provides a very fast reaction to a stimulus. To make this possible the stimulus is not subject to costly processing mechanisms of the brain but processed in a quite straightforward way. Beyond this simple mechanism the human *perception* is based on changing and interpreting sensory information. Zimbardo et al. [ZG99] split the process of cognition into

1. sensation,
2. organisation, and
3. identification and classification.

Sensation is the process of receiving and coding stimuli from the environment. Men have nine modalities for sensory experiences: seeing, hearing, smelling, tasting, touching, temperature sense, equilibrium sense, kinaesthesis, and sensation of pain. In every modality cognition starts with the detection of a stimulus by detectors. These transform the physical signal into a sensory signal which can be processed by the nervous system. A sensory threshold is the level at which a stimulus or a change of stimulus can be detected. There are two different kinds of thresholds: absolute thresholds and difference thresholds. An absolute threshold is the minimal level of intensity at which a physical energy causes a sensation. The smallest difference between two physical stimuli that can be discriminated is the difference threshold.

The organisation of the perception means the transformation of a stimulus into an interior representation. This includes the integration and combina-

tion of different sensory information like sound and shape into one percept of an object that can be recognised later on. In the last step, the identification and classification, the percepts are assigned to a meaning. This is a high level cognitive process that is based on the memories and the knowledge of a person. An example of a process of this step is to assign a thing of a certain shape, colour, sound, and so on to the category "car".

Another aspect affecting our process of perception is *attention*. It can be seen as a filter that helps to cope with the heavy load of information that are arriving all the time. It blocks the majority of this flood of data; only letting certain relevant information pass. This is very important as our brain has a limited processing capability that would be overburdened by the surplus of stimuli.

All creatures have a certain learning potential. This potential varies from species to species: some animals are dominated by inherent reactions on certain stimuli and learn rather few new things while the behaviour of others is less pre-assigned and they have a greater ability to learn. There are different models of learning, however all of them are based on experiences, i.e. the things that happen to an individual. Especially interesting types of learning for this work are inductive and observational learning as well as exchange of knowledge. Inductive learning is learning by example, where a general rule is induced from a set of observed instances. Observational learning refers to the ability to learn by the observation of another individuals' actions. The exchange of knowledge guarantees that the (learnt) knowledge of an individual is propagated for the benefit of many others. Observational learning and the exchange of knowledge are very efficient, and allow tedious and sometimes dangerous trial-and-error procedures of other types of learning to be avoided.

The *memory* gives us the ability to absorb, store, and recall information. It is a storage for knowledge of the world, like facts, beliefs, and so on. All of our cognitive processes are based on our memory. It can be separated into declarative and procedural memory. The declarative memory is a fact base and contains facts or events like names or the last visit to a soccer match. The procedural memory contains knowledge about how things are done and is used to acquire, store, and apply certain abilities like riding a bicycle. From an information-processing point of view the memory can be separated into the stages of sensory memory, short-term memory, and long-term memory. For every modality of sensation we have a sensory memory. It is a collection of sensory information after a stimulus has been received. The sensory memory is transient to guarantee that old sensations do not interfere with new sensations. However it is also persistent enough to provide some kind of continuity of the sensations. A common example to demonstrate the short-term memory is the looking up of a telephone number in a telephone book, holding it in memory long enough to dial the number but to forget

it afterwards. The short-term memory has a limited capacity which results from the human beings' inability to deal with every aspect of the environment that is surrounding us. Like the filtering property of attention the limited capacity of the short-term memory serves to focus our processing powers onto certain things. The recall of information from the short-term memory is very fast. It also serves as a working memory where information are edited, reconceived, and restructured. The information comes from the sensory or the long-term memory. The long-term memory represents the store of general knowledge, experiences, abilities, emotions, words and so on. Information stored there can often be recalled lifelong while its capacity is theoretically unlimited. It does not only contain our experiences but also information derived from these experiences together with plans for behaviour.

Problem solving and *reasoning* are central cognitive abilities. A problem consists of an initial state, a target state, and a set of operators. The initial state is a current unsatisfying state that should be transformed into a desired target state. The operators are the available possibilities to affect the environment. A well-defined problem has a clear definition of the initial state, the target state, as well as the available operations. If one of these three components is unclear the problem is called ill-defined. Reasoning is the process of drawing conclusions based on knowledge to solve problems, generate derived knowledge, and so on which is the basic feature that enables us to act intelligently and autonomically.

We have a lot of information about our environment which are uncertain but we need to make decisions anyway to act properly. Furthermore, often a fast but good enough action is better than an optimal but late action. These are issues we have to deal with every day and to cope with them goes beyond linear problem solving or reasoning: *judgement* and *decision-making*. Judgement is the process of forming an opinion, to draw a conclusion, and to rate existing information. Decision making refers the process that leads to the selection of a course of action among options. Judgement heuristics are important to generate efficient judgements and to make fast decisions as they reduce the set of possible choices. They allow us to act quickly despite our limited processing capacities and the limited information we have. But they also can lead to wrong assessments that could have been avoided using more expensive reasoning.

5.2.2 Sociological concepts

The psychological approach taken so far covers the characteristics of individuals. Sociology as an inter-individual discipline goes beyond that and provides concepts to analyse and explain properties of an aggregation of in-

dividuals. A social *role* can be seen as expectations that society places on the behaviour of an individual in a certain situation. A role typically includes rights and obligations and can change over time. 'Student' and 'professor' are examples of social roles - here in a university context. An individual that has the role 'professor' in one situation also can have the role 'husband' in another. These examples also demonstrate that roles can interfere with each other.

All social acting by individuals is influenced by *norms* which are specifications, instructions, or rules. They help the individual to behave appropriately and support the common welfare. They provide a scale of values for a society. A social *network* is a network of relationships between individuals, groups, or organisations. The relations can be based on spatial proximity, frequency of communication, friendship, financial exchange, similar ideas, and other forms of social interaction. Social networks often form a small-world structure where the maximal number of hops between its members is low. Social networks enjoy a growing research interest as they seem to adequately represent the relations of a person in many aspects.

The introduced concepts from psychology and sociology form the basis of the node manager that is presented in the following section.

5.3 Architecture

In this section the architecture of the node manager is presented. The node manager is a concept to autonomically control the nodes of a distributed system. It is supposed that there exists some kind of middleware that is running on every node to provide to the node manager a basic infrastructure to send and receive messages.

The node managers control the nodes in a distributed manner with no central instance. They act autonomically, ensuring the system is in a desired state defined by the administrator or user respectively. The intelligence to act properly in complex environments is inspired by psychological concepts. While there is no central instance of control, the node managers follow social communication, interaction patterns, roles, and norms to realise global targets.

The node managers are able to observe their environment through receptors or sensors that are able to sense certain types of stimuli. The data these sensors provide are called sensory information. An example of a certain kind of sensor is a failure detector - it provides information about the failure of other nodes. Further examples are temperature sensors as used in sensor networks, a sensor that informs the node managers about the avail-

able memory of a node, and so on. Basically all data or information reaching the node manager is sensory information. As a cooperative, distributed system is considered, communication with other node managers also form an important source of information. From an exterior point of view the node managers are a black box with sensory information as an input and actions as an output. The actions the node managers perform are intended to maintain the distributed system in a desired, user-defined state. If it is impossible to meet all desired conditions the actions should at least lead to graceful degradation.

The task of a node manager is to autonomically control applications or services running on the nodes. A typical software layer architecture is considered with an operating system hosting a middleware. Application services are running on the middleware as parts of user applications. The node manager depends on the middleware as it needs to communicate with other node managers, at least if non-local properties have to be achieved or maintained. It autonomically controls application services and the middleware and is designed to carry over responsibilities from the user or administrator to the node manager. The node manager has certain goals which are determined by the administrator while the preservation of these goals are left to the node managers. The PNS of the node manager is the connector between the environment and the CNS. It receives sensory information and transmits it to the CNS that processes it. If the CNS takes a decision to act, it is the PNS that triggers its actual execution. Figure 5.3 illustrates the described integration of the node managers into the software architecture.

In the following the particular functionality of the node manager is explained. Therefore its typical workflow as illustrated in Figure 5.4 is considered.

As described above, the starting point are stimuli or sensory information. At first, these pass a sensory filter which can be attributed to the PNS and only transmits sensory information of a certain defined form. This prevents useless information to even reach the CNS to minimise the waste of processing capabilities. After this barrier the stimulus has reached the CNS, more precisely the spinal cord. There it can trigger a reflex if a fast reaction is necessary. In this case the CNS immediately sends an instruction to the PNS which initiates its execution. Anyway the stimulus is transferred to the brain, where a component organises, identifies, and classifies it. The result of this process is stored into a memory component. The memory forms a knowledge base for the high-level capabilities of the node manager. A learning component enables it to generate new knowledge to improve performance, adaption, and so on. A decision-making component provides reasoning capabilities to the node manager. Thus it can autonomically generate plans for achieving the desired properties of the distributed system. The node manager has knowledge about social behaviour that affects partic-

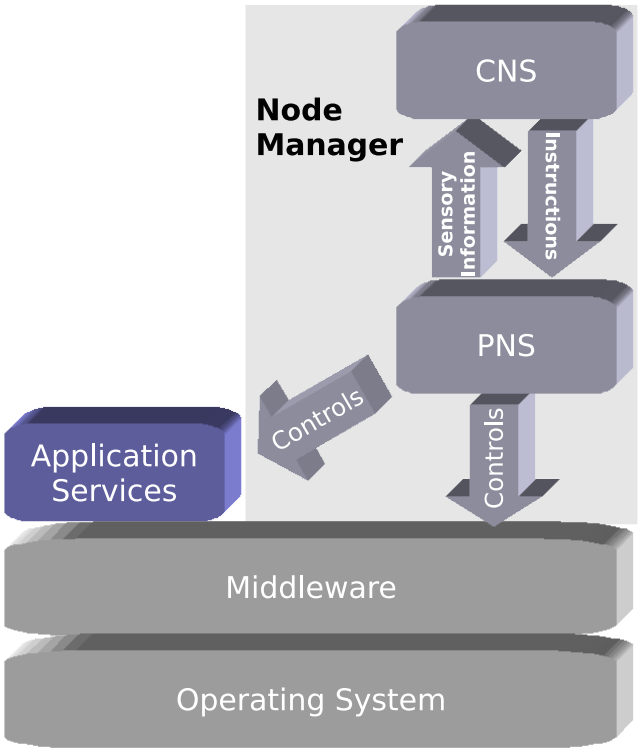


Figure 5.3: *Architectural overview of a node*

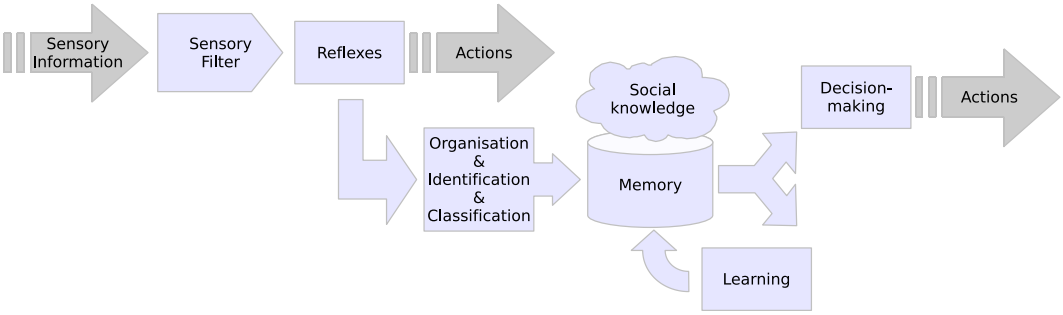


Figure 5.4: *Typical workflow of the Node Manager*

ularly the decision-making component and leads to coordination and cooperation with the other node managers. The execution of the planned actions is again left to the PNS.

Now all the main concepts of the node manager have been touched on. In the following they will be explained in detail.

5.3.1 Sensory filter

The node managers perceive their environment with sensors. The term sensor or sense refers in this work to every facility that contributes to gather information about external or internal states. This could be a real hardware temperature sensor but also a sensor that provides information about the current workload of the node or about the state of some application or service. It is assumed that there are two kinds of sensors, depending on whether they provide continuous or discrete values. Let $\mathscr{S}$ be the set of existing senses, and $\mathscr{P}$ be the corresponding set of possible perceptions. A function $\rho : \mathscr{S} \rightarrow \mathscr{P}$ maps a sensor to its corresponding value set which is either discrete or continuous. A temperature sensor $S_t \in \mathscr{S}$ could for example be matched to the value set $[-10C^\circ, 70C^\circ] \in \mathscr{P}$ if it provides values within this range.

Be $\mathcal{P} \in \mathscr{P}$ the possible percepts of an arbitrary sensor. The sensory filter function $\varphi_\mathcal{P} : \mathcal{P} \rightarrow \{0,1\}$ determines which perceptions are discarded and which are further processed. The perceptions $\mathcal{P}_\varphi \subseteq \mathcal{P}$ that are not discarded and pass the filter are defined as follows: $\mathcal{P}_\varphi := \{x \in \mathcal{P} \mid \varphi_\mathcal{P}(x) = 1\}$. A simple filter function for the temperature sensor could be:

$$\varphi_{P_t}(x) = \begin{cases} 1 & \text{if } x \geq 0C^\circ \\ 0 & \text{if } x < 0C^\circ \end{cases}$$

where temperatures smaller than $0C^\circ$ are filtered out. Also, percepts from the percept history can be used in the filter if the sensory memory contains them. In this way, filters similar to relative threshold filters can be defined, e.g. to only let pass temperature perceptions that have changed more than $1C^\circ$ compared to the last percepted temperature x_{-1}:

$$\varphi_{P_t}(x) = \begin{cases} 1 & \text{if } |x - x_{-1}| > 1C^\circ \\ 0 & \text{otherwise.} \end{cases}$$

5.3.2 Reflexes

A reflex is a concept that allows very fast reactions to perceptions after having passed the sensory filter. Reflexes are modelled by simple rules of the form `IF condition THEN action` what represents a reflex arc. For the specification and processing of these rules a rule engine like JESS[1] might be used.

A rule of a rule engine is quite similar to an `IF ... THEN` statement of a program language. However, rule engines can be much more efficient. JESS

[1] http://herzberg.ca.sandia.gov/jess/

for instance is using the RETE [For82] algorithm which can be many orders of magnitude faster than equivalent `IF ... THEN` statements.

Reflexes are defined as follows:

```
reflex REFLEXNAME
  (CONDITIONS) =>
  (ACTIONLIST)
```

`REFLEXNAME` is just a name for the reflex. The `CONDITIONS` define under which conditions the actions of the `ACTIONLIST` are fired. Each element of the `ACTIONLIST` is an element of $\mathcal{A}$. The `CONDITIONS`, the facts the reflex rules operate on, are based on the filtered percepts $\mathcal{P}_\varphi$ of the sensors of the node manager. They have the same structure as the preconditions of actions of a planning language, as introduced in the previous chapter.

5.3.3 Organisation, identification, and classification of stimuli

Organisation, identification, and classification is necessary to transform the stimuli into a representation suitable for further processing. This transformation is also conducted with rules. These have the form:

```
oic OICNAME
  (CONDITIONS) =>
  (FACTS)
```

The term `CONDITIONS` is again the precondition that has to be true for the rule to be executed. However, the rules produce facts that are stored in the memory, more precisely the short term memory, instead of triggering actions. To get a better impression on how the OIC-rules are used, consider the following example:

```
oic weather
  (temperature in [15, 25] &&
   humidity < 80% &&
   wind < 50 km/h) ?
   => (goodweatherconditions) :
   => (NOT goodweatherconditions)
```

The rule classifies the weather conditions into the categories or concepts of good and not good dependent on temperature, humidity, and wind. If this

information is sufficient for decision-making then this classification represents a huge reduction of complexity for this process. The set of percepts $\mathscr{P}$ is transformed by the OIC-rules to a set of facts $\mathscr{F}$.

5.3.4 Memory

The memory is separated into sensory memory, short term memory, and long term memory. The sensory memory is modelled by queues of a certain size, one for each sense. If only the latest value is of interest the size can be set to one, otherwise it is set to a certain higher value. After every sensing cycle the actual values are inserted into the sensory memory.

The short term memory stores recognised concepts, facts, and so on. Information which is accessed frequently, or other impressive and essential information, is moved to the long-term memory. The main difference between these two types of memory is the storage time of knowledge.

In literature, the long term memory is further divided into several categories. The most interesting subcategories are probably the semantic and the episodic memory. The semantic memory stores knowledge like 'humans have two legs' while the episodic consists of stored experiences.

An adequate storage and retrieval of information is very important for decision-making and learning. The memory component is only sketched as it is beyond the scope of this work to derive a fine-grained model.

5.3.5 Decision-making

Each node manager has a number of actions it is able to perform. Decision-making is choosing between such alternatives. The core of this component for the node managers is a reasoning engine, for instance an automated planner as proposed in the previous chapter. Due to the simplification of the stimuli by the organisation, identification, and classification component, and learning progress, the reasoning becomes more efficient. As a planning engine is presented in detail in the previous chapter, a further explanation of basic reasoning concepts is omitted here.

It is desirable that a decision-making component follows the idea of anytime execution, i.e. a solution is available at anytime, but the more time that is available, the more accurate the processing performance will be.

5.3.6 Learning

The consequences of past decisions of the decision-making component are used as feedback to enable learning from past situations. Especially interesting types of learning for this work are inductive and observational learning as well as exchange of knowledge. Inductive learning is learning by example, where it is tried to induce a general rule from a set of observed instances. Observational learning refers to the ability to learn by the observation of other individuals' actions.

As distributed systems consists of many node managers, observational, cooperative learning and exchange of experiences are valuable. Thus, if a node manager is able to derive a successful behaviour all node managers benefit from that.

5.3.7 Cooperation

The cooperation among node managers is necessary to fulfil the system's goals and to guarantee scalability. Because of limited processing power, memory, and information, a central node manager is not able to control a complex distributed system. Social knowledge is necessary to enable a reasonable cooperation of the node managers. Furthermore, the social knowledge comprises the goals and requirements of the whole system which have to be maintained by the node managers. This is encoded as common norms and supports the "welfare" of the distributed system. All acting is influenced by these norms.

Roles are adopted by and distributed among the node managers to ensure a scalable assignment of things such as tasks which need to be performed. The relations between the node managers are mapped to social networks. For very complex systems higher level networks are possible, i.e. a network consisting of networks. To give an idea of the benefits of such an organisation consider a node manager needing assistance to execute an action, it can start to ask for this assistance in its local network first and broadens the request gradually. The closer other entities are to the node manager the more information it should know about it. With this functionality it should be possible to keep communication locally and to distribute information intelligently. The grouping algorithms of Chapter 3 could provide some fundamental capability to form such social networks. As a global understanding of the node managers is necessary for communication and cooperation and the node managers do not necessarily share the same implementation, an abstraction of the virtual environment is needed. For this purpose an ontology can be used. An ontology is a formal specification of a set of concepts and its relationships.

5.4 Conclusions

In this concluding chapter, an architecture for self-healing in highly complex distributed systems is sketched. The design of the architecture is inspired by psychological and sociological aspects. Many of the techniques presented in the previous chapters could be taken up in this architecture. The planning engine presented in Chapter 4 could serve as a basis of the architecture's decision-making component. The cooperation of the node managers based on social networks could be built upon the grouping algorithms proposed in Chapter 3 which could be used to form such groups. In that case, the suitability of two nodes which influences the grouping algorithms should be defined based on social aspects. The node manager's perception of the environment depends on sophisticated monitoring information. In distributed environments it is especially tough to detect node crashes. To be able to detect the failure of other node managers, monitoring input of a component, like the failure detector of Chapter 2, is needed. Thus, this chapter provides an overview of possible future research in the struck way of self-healing systems.

There are two kinds of people, those who finish what they start and so on.

Robert Byrne

6 Conclusions

This dissertation dealt with approaches to enable self-healing distributed systems. Therefore, generic key issues have been identified and methods of resolution have been provided.

Commonly accepted models of systems with self-x features suggest that the main architectural blocks are an observing and a controlling instance. For self-healing, the main tasks of observation and control are the detection of flaws and their correction.

The detection of node failures is a generic, non-trivial task in distributed environments. In this work a new failure detection algorithm has been proposed. It is flexible, adaptive, and has outperformed all other considered algorithms in the evaluated settings. Different variants of the basic detection algorithm have been investigated and further improvements could be achieved. The evaluations suggest that the developed algorithm is the best heartbeat-style failure detector if message loss is considered. In some cases the algorithm proposed in this work makes about 90% fewer wrong suspicions than other algorithms. Furthermore, a method called lazy monitoring has been developed to reduce the overhead of failure detectors which depend on heartbeat messages. Besides the reduction of overhead the lazy monitoring also contains the possibility of a faster adaption to changing network conditions and better detection quality. In the considered testbed the usage of lazy monitoring reduced the traffic to 1.2% and the number of messages to 1% of the benchmark. At the same time 10 times more information about the environment could be gathered what helps failure detectors to perform better.

To allow for scalable systems which have the ability to monitor themselves, the problem of an autonomous installation of monitoring relations in distributed systems has been investigated. This novel problem has been defined in a concise way and three algorithms to solve this problem have been derived. They have been compared regarding their efficiency, suitability, and the failure tolerance they are providing. The evaluation shows that the overhead of all proposed algorithms is independent from the network size and therefore suitable for complex large scale distributed system. Each algorithm needs only a very limited number of messages per node in order to fully install monitoring relations.

A failure recovery engine has been developed to manage distributed systems and recover them from unwanted states. It is a generic approach applicable to any scenario whose features can be represented by the provided modelling language. Humans only need to specify the desired system properties while it is left to the engine to reach them. Thus, responsibilities are shifted from users and administrators to the system itself. The failure recovery engine is reliable and trustworthy as it is based on a sound and complete automated planning approach. Therefore it contributes to cope with more and more complex systems and minimises human error. To identify unwanted system states, a technique allowing a distributed consistency check has been introduced. A distributed planning algorithm called DPOP has been developed. A plan execution method provides the ability to concurrently execute plans in distributed systems. Several extensions are presented which aim to provide extra functionality and better performance. The functionality of the proposed approach is evaluated within different scenarios. These provide a proof of concept that the approach is suitable to autonomically manage quite complex systems with more than 100 nodes. An introduced technique called recovery-oriented planning resulted in saving up to 97% of planning steps, especially if only small parts of a system fail.

One chapter has been dedicated to sketch an architecture to manage highly complex distributed systems as starting point for future work. It is inspired by psychology and sociology and tries to adapt concepts which enable humans to cope with its highly complex environment to technical systems. Many of the techniques introduced in the main chapters could be integrated in such an architecture.

Bibliography

[ACE+03] R. Allrutz, C. Cap, S. Eilers, D. Fey, H. Haase, C. Hochberger, W. Karl, B. Kolpatzik, J. Krebs, F. Langhammer, P. Lukowicz, E. Maehle, J. Maas, C. Müller-Schloer, R. Riedl, B. Schallenberger, V. Schanz, H. Schmeck, D. Schmid, W. Schröder-Preikschat, T. Ungerer, H.-O. Veiser, and L. Wolf. Organic Computing - Computer- und Systemarchitektur im Jahr 2010 (in German), 2003. VDE/ITG/GI position paper.

[AHT90] James Allen, James Hendler, and Austin Tate, editors. *Readings in Planning*. Morgan Kaufmann, San Mateo, CA, 1990.

[AHW04] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. A planning based approach to failure recovery in distributed systems. In David Garlan, Jeff Kramer, and Alexander L. Wolf, editors, *WOSS*, pages 8–12. ACM, 2004.

[ALR04] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats - A taxonomy. In René Jacquart, editor, *IFIP Congress Topical Sessions*, pages 91–120. Kluwer, 2004.

[And98] Jesper Andersson. Reactive dynamic architectures. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 1–4, New York, NY, USA, 1998. ACM.

[And08] Thorge Andersen. Automatisches Planen für selbstheilende verteilte Systeme (in German). Diploma thesis, University of Augsburg, 2008.

[ARLV01] C. Asavathiratham, S. Roy, B. Lesieutre, and G. Verghese. The influence model. *Control Systems Magazine, IEEE*, 21(6):52–64, Dec 2001.

[Bar96] Valmir C. Barbosa. *An introduction to distributed algorithms*. MIT Press, Cambridge, MA, USA, 1996.

[BF95] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, August 1995.

[BMMSP06] J. Branke, M. Mnif, C. Müller-Schloer, and H. Prothmann. Organic computing addressing complexity by controlled self-organization. *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 185–191, Nov. 2006.

[BMS02] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 354–363, Washington, DC, USA, 2002. IEEE Computer Society.

[BMS03] Marin Bertier, Olivier Marin, and Pierre Sens. Performance analysis of a hierarchical failure detector. In *Proceedings 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 635–644, San Francisco, CA, USA, June 2003. IEEE Computer Society.

[Bol93] Jean-Chrysotome Bolot. End-to-end packet delay and loss behavior in the internet. In *SIGCOMM '93: Conference proceedings on Communications architectures, protocols and applications*, pages 289–298, New York, NY, USA, 1993. ACM.

[BS72] B. S. Baker and R. Shostak. Gossips and telephones. *Discrete Math.*, 2(3):191–193, June 1972.

[Car88] J. G. Carbonell. PRODIGY: an integrated architecture for planning and learning. In *Symposium for methodologies for intelligent systems 3*, Turin, October 1988.

[CASD85] F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, pages 200–206, Ann Arbor, MI, USA, 1985. IEEE Computer Society Press.

[CDK00] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, third edition, 2000.

[Cha87] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[CTA00] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. In *Proceedings of the In-*

ternational Conference on Dependable Systems and Networks (DSN 2000), New York, 2000. IEEE Computer Society Press.

[DGH+87] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, and John Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, Vancouver, British Columbia, Canada, 10–12 August 1987.

[DGM02] Abhinandan Das, Indranil Gupta, and Ashish Motivala. SWIM: Scalable weakly-consistent infection-style process group membership protocol. In *DSN '02: 2002 International Conference on Dependable Systems and Networks*, pages 303–312. IEEE Computer Society, 2002.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[DN98] Kaushik K. Dam and Lionel M. Ni. Design and implementation of a network emulator. Technical Report MSU-CPS-ACS-98-16, Department of Computer Science and Engineering, Michigan State University, May, 1998.

[DS82] D. Dolev and H. R. Strong. Distributed commit with bounded waiting. In *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh*, pages 53–59. IEEE, July 1982.

[EH04] Stefan Edelkamp and Joerg Hoffmann. PDDL2.2: The language for the classical part of the 4th international planning competition. report00195, Institut für Informatik, Universität Freiburg, January 21 2004.

[EMW97] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic sat-compilation of planning problems. In *In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176. Morgan Kaufmann, 1997.

[FDGO99] P. Felber, X. Défago, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, pages 132–141, Edinburgh, Scotland, 1999.

[Fel70] W. Feller. *An Introduction to Probability Theory and its Application, Vol. 1.* John Wiley and Sons, New York, 1970.

[FL98] Maria Fox and Derek Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.

[FL01] Maria Fox and Derek Long. STAN4: A hybrid planning strategy based on subproblem abstraction. *The AI Magazine*, 22(1):81–84, 2001.

[FL03] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[Fly72] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, September 1972.

[FN71] R. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence-4, 1971*, 2:189–208, 1971.

[For82] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[Fox02] Armando Fox. Toward recovery-oriented computing. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases*, pages 873–876. VLDB Endowment, 2002.

[FRT01] Christol Fetzer, Michel Raynal, and Frederic Tronel. An adaptive failure detection protocol. In *PRDC '01: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing*, page 146, Washington, DC, USA, 2001. IEEE Computer Society.

[FvR97] Roy Friedman and Robbert van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *Symp. on High Performance Distributed Computing, HPDC*, pages 233–242, 1997.

[GAKM02] Gillen, Al, Dan Kusnetzky, and Scott McLaron. The role of linux in reducing the cost of enterprise computing, 2002. IDC white paper.

[GCG01] Gupta, Chandra, and Goldszmidt. On scalable and efficient distributed failure detectors. In *PODC: 20th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2001.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[GL05] Alfonso Gerevini and Derek Long. Plan constraints and preferences in pddl3. Technical report, Department of Electronics for Automation, University of Brescia, August 2005.

[GM82] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, January 1982.

[GNT04] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufman, San Francisco, CA, 2004.

[GOR06] Matthias Güdemann, Frank Ortmeier, and Wolfgang Reif. Formal modeling and verification of systems with self-x properties. In Laurence Tianruo Yang, Hai Jin, Jianhua Ma, and Theo Ungerer, editors, *ATC*, volume 4158 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2006.

[Gre69] Cordell Green. Application of theorem proving to problem solving. pages 219–239. Morgan Kaufmann, 1969.

[GS96] Alfonso Gerevini and Lenhart Schubert. Accelerating partial-order planners: Some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research*, 5:95–137, 1996.

[GSRU07] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu J. Upadhyaya. Self-healing systems - survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007.

[Här91] W. Härdle. *Smoothing Techniques with Implementation in S.* Springer Verlag, Berlin, 1991.

[Has00] Patrik Haslum. Admissible heuristics for optimal planning. pages 140–149. AAAI Press, 2000.

[HDYK04] Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. The f accrual failure detector. In *23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, pages 66–78. IEEE Computer Society, 2004.

[HK97] Markus Horstmann and Mary Kirtland. Dcom architecture. Technical report, http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomarch.htm, July 1997.

[HNR72] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bulletin*, (37):28–29, 1972.

[Hof01] Jörg Hoffman. FF: The fast-forward PLanning system. *The AI Magazine*, 22(1):57–62, 2001.

[Hor01] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology. *http://www.research.ibm.com/autonomic/*, 2001.

[HTC05] Yuuki Horita, Kenjiro Taura, and Takashi Chikayama. A scalable and efficient self-organizing failure detector for grid applications. In *SC'05: Proc. The 6th IEEE/ACM International Workshop on Grid Computing CD*, pages 202–210, Seattle, Washington, USA, November 2005. IEEE/ACM.

[IBM06] IBM. An architectural blueprint for autonomic computing, Jun 2006. White Paper, 4th edition.

[Jac88] V. Jacobson. Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, pages 314–329, New York, NY, USA, 1988. ACM Press.

[JP94] David Joslin and Martha E. Pollack. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *AAAI*, pages 1004–1009, 1994.

[JSHS05] Kaustubh R. Joshi, William H. Sanders, Matti A. Hiltunen, and Richard D. Schlichting. Automatic model-driven recovery in distributed systems. In *Proceedings 24th IEEE Symposium on Reliable Distributed Systems (24th SRDS'05)*, pages 25–38, Orlando, FL, USA, October 2005. IEEE Computer Society.

[KH00] Jana Koehler and Jörg Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, 12:338–386, 2000.

[KNHD97] Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. Extending planning graphs to an adl subset. In *ECP '97: Proceedings of the 4th European Conference on Planning*, pages 273–285, London, UK, 1997. Springer-Verlag.

[Koe98] Jana Koehler. Planning under resource constraints. In *European Conference on Artificial Intelligence*, pages 489–493, 1998.

[Kor88] R. E. Korf. *Search: A survey of recent results*, pages 197–237. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[KS86] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[KS92] Henry Kautz and Bart Selman. Planning as satisfiability. In *ECAI '92: Proceedings of the 10th European conference on Artificial intelligence*, pages 359–363, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[KS96] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *AAAI/IAAI, Vol. 2*, pages 1194–1201, 1996.

[KS98] Henry Kautz and Bart Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Workshop on Planning as Combinatorial Search, in conjunction with AIPS-98 (Conference on Artificial Intelligence Planning Systems*, pages 58–60, 1998.

[KS99] Henry Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In Thomas Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 318–325, San Francisco, 1999. Morgan Kaufmann.

[KWWW94] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A note on distributed computing. Technical report, Mountain View, CA, USA, 1994.

[LAF99] Mikel Larrea, Sergio Arévalo, and Antonio Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 34–48, London, UK, 1999. Springer-Verlag.

[LF00] Derek Long and Maria Fox. Automatic synthesis and use of generic types in planning. In *Artificial Intelligence Planning Systems*, pages 196–205, 2000.

[Lib00] Paolo Liberatore. On the complexity of choosing the branching literal in DPLL. *Artificial Intelligence*, 116(1–2):315–326, 2000.

[Lif86] Vladimir Lifschitz. On the semantics of STRIPS. In Michael P. Georgeff and Amy Lansky, editors, *Reasoning about Actions and Plans*, pages 1–9. Morgan Kaufmann, Los Altos, California, 1986.

[LLS+07] Mikel Larrea, Alberto Lafuente, Iratxe Soraluze, Roberto Cortiñas, and Joachim Wieland. On the implementation of communication-optimal failure detectors. In *3rd Latin-American Symp. on Dependable Computing, LADC*, volume 4746 of *LNCS*, pages 25–37, Morelia, Mexico, 2007. Springer Verlag.

[Lyn89] N. Lynch. A hundred impossibility proofs for distributed computing. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 1–28, New York, NY, USA, 1989. ACM Press.

[McC68] John McCarthy. Situations, actions and causal laws. In M. Minsky, editor, *Semantic Information Processing*, pages 410–417. MIT Press, 1968.

[McC86] John McCarthy. Applications of circumscription to formalizing common sense reasoning. *Artificial Intelligence*, 28:89–116, 1986.

[McD98] D. McDermott. Pddl — the planning domain definition language, 1998.

[MR91] David A. McAllester and David Rosenblitt. Systematic nonlinear planning. In *AAAI*, pages 634–639, 1991.

[MS99] Rob Miller and Murray Shanahan. The event calculus in classical logic - alternative axiomatisations. *Electron. Trans. Artificial Intelligence*, 3(A):77–105, 1999.

[Muk92] Amarnath Mukherjee. On the dynamics and significance of low frequency components of internet load. Technical Report MIS-CIS-92-83, University of Pennsylvania, December, 1992.

[NFF+05] Alexander Nareyek, Eugene C. Freuder, Robert Fourer, Enrico Giunchiglia, Robert P. Goldman, Henry Kautz, Jussi Rintanen, and Austin Tate. Constraints and ai planning. *IEEE Intelligent Systems*, 20(2):62–72, 2005.

[NK01] XuanLong Nguyen and Subbarao Kambhampati. Reviving partial order planning. In Bernhard Nebel, editor, *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, pages 459–466, San Francisco, CA, August 4–10 2001. Morgan Kaufmann Publishers, Inc.

[NS72] A. Newell and H. A. Simon. *Human problem solving*. Prentice-Hall, 1972.

[OGT+99] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE [see also IEEE Intelligent Systems]*, 14(3):54–62, May/Jun 1999.

[Pap84] A. Papoulis. *Probability, Random Variables and Stochastic Processes*. McGraw-Hill, second edition, 1984.

[Pat02] David Patterson. Embracing failure: Recovery oriented computing (ROC). In *The Conference on High Speed Computing*, page 9, Salishan Lodge, Gleneden Beach, Oregon, April 2002. LANL/LLNL/SNL. LA-UR-02-2865.

[PBB+02] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman,

and Noah Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques,. Technical report, March 26 2002.

[Ped89] Edwin P. D. Pednault. Adl: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[PJP97] Martha E. Pollack, David Joslin, and Massimo Paolucci. Flaw selection strategies for partial-order planning. *Journal of Artificial Intelligence Research*, 6:223–262, 1997.

[PRT+08] Holger Prothmann, Fabian Rochner, Sven Tomforde, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck. Organic control of traffic lights. In Chunming Rong, Martin Gilje Jaatun, Frode Eika Sandnes, Laurence Tianruo Yang, and Jianhua Ma, editors, *ATC*, volume 5060 of *Lecture Notes in Computer Science*, pages 219–233. Springer, 2008.

[PS93] Mark A. Peot and David E. Smith. Threat-removal strategies for partial-order planning. In *AAAI*, pages 492–499, 1993.

[PTC06] Barry Porter, Francois Taiani, and Geoff Coulson. Generalised repair for overlay networks. In *SRDS '06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 132–142, Washington, DC, USA, 2006. IEEE Computer Society.

[PW92] J. S. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning (KR-92)*, pages 103–114, October 1992.

[PW94] J. Scott Penberthy and Daniel S. Weld. Temporal planning with continuous change. In *AAAI*, pages 1010–1015, 1994.

[R D05] R Development Core Team. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005.

[Rei01] Ray Reiter. On knowledge-based programming with sensing in the situation calculus. *ACM Transactions on Computational Logic (TOCL)*, 2(4):433–457, October 2001.

[RMB+06] Urban Richter, Moez Mnif, Jürgen Branke, Christian Müller-Schloer, and Hartmut Schmeck. Towards a generic observer/controller architecture for organic computing. In Christian Hochberger and Rüdiger Liskowsky, editors, *INFORMATIK 2006 – Informatik für Menschen*, volume P-93 of *GI-*

Edition – Lecture Notes in Informatics, pages 112–119, Bonn, Germany, September 2006. Köllen Verlag.

[RMH98] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. Technical Report TR98-1687, Cornell University, Computer Science, May 28, 1998.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[Ros81] S. J. Rosenschein. Plan synthesis: A logical perspective. In *Proceedings 7th IJCAI*, pages 331–337, 1981.

[RWS06] Sandip Roy, Yan Wan, and Ali Saberi. *Algorithmic Aspects of Wireless Sensor Networks*, volume 4240/2006 of *LNCS*, chapter A Flexible Algorithm for Sensor Network Partitioning and Self-partitioning Problems, pages 152–163. Springer Berlin / Heidelberg, 2006.

[Sac74] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[Sac90] Earl D. Sacerdoti. The nonlinear nature of plans. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 162–170. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1990.

[SAGJ93] Dheeraj Sanghi, Ashok K. Agrawala, Olafur Gudmundsson, and Bijendra N. Jain. Experimental assessment of end-to-end behavior on internet. In *INFOCOM '93: Proceedings of the Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future*, pages 867–874, 1993.

[SG95] Lenhart Schubert and Alfonso Gerevini. Accelerating partial order planners by improving plan and goal choices. In *In Proceedings of the Seventh International Conference on Tools with Artificial Intelligence*, pages 442–450. IEEE Computer Society Press, 1995.

[Sil86] B. W. Silverman. Kernel density estimation technique for statistics and data analysis. In *Monographs on statistics and applied probability*, volume 26. Chapman and Hall, London, 1986.

[SKC95] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *AAAI-92: Proceedings 10th National Conference on AI*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, January 1995.

[SPTU05] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer

Unland. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, 19(3):181–187, 2005.

[SPTU07a] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *SAC 2007: Proceedings of the 22nd ACM symposium on Applied computing*, pages 551–555, New York, NY, USA, 2007. ACM.

[SPTU07b] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. Variations and evaluations of an adaptive accrual failure detector to enable self-healing properties in distributed systems. In *ARCS 2007: Proceedings of the 20th International Conference on Architecture of Computing Systems*, volume 4415 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2007.

[SPTU08a] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A lazy monitoring approach for heartbeat-style failure detectors. In *ARES 2008: Proceedings of the 3rd IEEE International Conference on Availability, Reliability and Security*, IEEE Transactions, pages 404–409. IEEE Computer Society, 2008.

[SPTU08b] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. Using automated planning for trusted self-organising organic computing systems. In *ATC 2008: Proceedings of the 5th International Conference on Autonomic and Trusted Computing*, volume 5060 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2008.

[Sri05] Biplav Srivastava. The case for automated planning in autonomic computing. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 331–332, Washington, DC, USA, 2005. IEEE Computer Society.

[ST96] John Slaney and Sylvie Thiebaux. Linear time near-optimal planning in the blocks world. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1208–1214, Portland, Oregon, USA, August 1996. AAAI Press / The MIT Press.

[SU08] Benjamin Satzger and Theo Ungerer. Grouping algorithms for scalable self-monitoring distributed systems. In *Autonomics 2008: Proceedings of the 2nd ACM/ICST International Conference on Autonomic Computing and Communication*, 2008.

[Sus75] Gerald Jay Sussman. *A Computer Model of Skill Acquisition*. American Elsevier Publishing Co., New York, 1975.

[SW99] David E. Smith and Daniel S. Weld. Temporal planning with mutual exclusion reasoning. In Dean Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol1)*, pages 326–337, S.F., July 31–August 6 1999. Morgan Kaufmann Publishers.

[Tat77] A. Tate. Generating project networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 888–893, 1977.

[TBPU03a] Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer. Smart Doorplate. In *First International Conference on Appliance Design (1AD)*, pages 24–28, Bristol, GB, May 2003. Reprinted in Personal Ubiquitous Computing (2003) 7: 221-226.

[TBPU03b] Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer. Smart doorplate. *Personal and Ubiquitous Computing*, 7(3-4):221–226, March 2003.

[TBPU05] Wolfgang Trumler, Faruk Bagci, Jan Petzold, and Theo Ungerer. Amun - autonomic middleware for ubiquitous environments applied to the smart doorplate. In *Advanced Engineering Informatics*, volume 19, pages 243–252, Washington, DC, USA, 2005. ELSEVIER.

[TEP+07] Wolfgang Trumler, Jörg Ehrig, Andreas Pietzowski, Benjamin Satzger, and Theo Ungerer. A distributed self-healing data store. In *Autonomic and Trusted Computing, 4th International Conference, ATC 2007, Hong Kong, China, July 11-13, 2007, Proceedings*, volume 4610 of *Lecture Notes in Computer Science*, pages 458–467. Springer, 2007.

[Tru06] Wolfgang Trumler. *Organic Ubiquitous Middleware*. PhD thesis, Universität Augsburg, July 2006.

[Tuk77] J. Tukey. *Exploratory Data Analysis*. Addison-Wesley, Reading, MA, 1977.

[TvS02] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[VR81] Yaakov L. Varol and Doron Rotem. An algorithm to generate all topological sorting arrangements. *The Computer Journal*, 24(1):83–84, 1981.

[VR99] Vincent Vidal and Pierre Régnier. Total order planning is more efficient than we thought. In *AAAI/IAAI*, pages 591–596, 1999.

[WAS98] Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending graphplan to handle uncertainty & sensing actions. In *AAAI/IAAI*, pages 897–904, 1998.

[Wel94] Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[Wil88] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1988.

[WW99] Steven A. Wolfman and Daniel S. Weld. The LPSAT engine and its application to resource planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 310–316, Stockholm, Sweden, July 31-August 6 1999. Morgan Kaufmann.

[YS03a] Håkan L. S. Younes and Reid G. Simmons. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, 20:405–430, 2003.

[YS03b] Håkan L. S. Younes and Reid G. Simmons. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, 20:405–430, 2003.

[ZG99] Philip G. Zimbardo and Richard J. Gerrig. *Psychologie*. Springer, Berlin, 7th edition, 1999.

List of Figures

List of Tables

List of Algorithms

Benjamin Satzger

Curriculum Vitae

Personal Information

Date of birth	May 3, 1979
Place of birth	Kaufbeuren, Bavaria, Germany
Nationality	German
Marital status	Married, one child

Education

2006–present	**Research and teaching assistant**, *Systems and Networking*, Department of Computer Science, University of Augsburg.
2005	**Diploma in Computer Science**, *University of Augsburg*, Degree: Dipl.-Inf. (comparable to Master's degree), final grade: "very good" – A.
2005	**Computer science studies**, *University of Valencia*, Spain.
2001–2005	**Computer science studies**, *University of Augsburg*, Germany.
1999–2001	**Professional training**, *Hirschvogel Automotive Group*, Software engineer graduation.
1998–1999	**Civilian service**, *AWO Bildungsstätte*, Pforzen.
1989–1998	**Grammar school**, *Jakob-Brucker-Gymnasium*, Kaufbeuren, with graduation.
1985–1989	**Basic education**, *VS Stöttwang-Westendorf*.

First Author Publications

Benjamin Satzger and Theo Ungerer. Grouping algorithms for scalable self-monitoring distributed systems. In *Autonomics 2008: Proceedings of the 2nd ACM/ICST International Conference on Autonomic Computing and Communication*, 2008.

Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. Using automated planning for trusted self-organising organic computing systems. In *ATC 2008: Proceedings of the 5th International Conference on Autonomic and Trusted Computing*, volume 5060 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2008.

Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A lazy monitoring approach for heartbeat-style failure detectors. In *ARES 2008: Proceedings of the 3rd IEEE International Conference on Availability, Reliability and Security*, IEEE Transactions, pages 404–409. IEEE Computer Society, 2008.

Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. Variations and evaluations of an adaptive accrual failure detector to enable self-healing properties in distributed systems. In *ARCS 2007: Proceedings of the 20th International Conference on Architecture of Computing Systems*, volume 4415 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 2007.

Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *SAC 2007: Proceedings of the 22nd ACM symposium on Applied computing*, pages 551–555, New York, NY, USA, 2007. ACM.

Benjamin Satzger, Markus Endres, and Werner Kießling. A preference-based recommender system. In *EC-Web 2006: Proceedings of the 7th International Conference on E-Commerce and Web Technologies*, volume 4082 of *Lecture Notes in Computer Science*, pages 31–40. Springer, 2006.

Further Publications

Faruk Bagci, Florian Kluge, Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. Experiences with a smart office project. In Laurence T. Yang, editor, *Mobile Intelligence: Mobile Computing and Computational Intelligence*. John Wiley & Sons, Inc., 2008.

Wolfgang Trumler, Markus Helbig, Andreas Pietzowski, Benjamin Satzger, and Theo Ungerer. Self-configuration and self-healing in AUTOSAR. In *14th Asia Pacific Automotive Engineering Conference*, Hollywood, California, USA, August 2007. SAE International.

Wolfgang Trumler, Jörg Ehrig, Andreas Pietzowski, Benjamin Satzger, and Theo Ungerer. A distributed self-healing data store. In *Autonomic and Trusted Computing, 4th International Conference, ATC 2007, Hong Kong, China, July 11-13, 2007, Proceedings*, volume 4610 of *Lecture Notes in Computer Science*, pages 458–467. Springer, 2007.

Wolfgang Trumler, Andreas Pietzowski, Benjamin Satzger, and Theo Ungerer. Adaptive self-optimization in distributed dynamic environments. In Giovanna Di Marzo Serugendo, Jean-Philippe Martin-Flatin, Mark Jélasity, and Franco Zambonelli, editors, *First IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, pages 320–323, Cambridge, Boston, Massachussets, 2007. IEEE Computer Society.

Andreas Pietzowski, Benjamin Satzger, Wolfgang Trumler, and Theo Ungerer. Using positive and negative selection from immunology for detection of anomalies in a self-protecting middleware. In *Informatik 2006, Informatik für Menschen*, volume P-93, pages 161–168, Dresden, Germany, October 2006. Gesellschaft für Informatik e.V., LNI.

M. Güdemann, F. Nafz, A. Pietzowski, W. Reif, B. Satzger, H. Seebach, and T. Ungerer, editors. Applications and architectures in Organic Computing (DFG SPP 1183 Organic Computing). Technical report, Department of Computer Science, University of Augsburg, 2006.

Andreas Pietzowski, Benjamin Satzger, Wolfgang Trumler, and Theo Ungerer. A bio-inspired approach for self-protecting an organic middleware with artificial antibodies. In *Self-Organising Systems, First International Workshop (IWSOS 2006)*, volume 1, pages 202–215, Passau, Germany, September 2006. Springer.